Asking and Answering Questions during a Programming Change Task

Jonathan Sillito, *Member*, *IEEE*,
Gail C. Murphy, *Member*, *IEEE*, and Kris De Volder

Abstract—Little is known about the specific kinds of questions programmers ask when evolving a code base and how well existing tools support those questions. To better support the activity of programming, answers are needed to three broad research questions: 1) What does a programmer need to know about a code base when evolving a software system? 2) How does a programmer go about finding that information? 3) How well do existing tools support programmers in answering those questions? We undertook two qualitative studies of programmers performing change tasks to provide answers to these questions. In this paper, we report on an analysis of the data from these two user studies. This paper makes three key contributions. The first contribution is a catalog of 44 types of questions programmers ask during software evolution tasks. The second contribution is a description of the observed behavior around answering those questions. The third contribution is a description of how existing deployed and proposed tools do, and do not, support answering programmers' questions.

Index Terms—Change tasks, software evolution, empirical study, development environments, programming tools, program comprehension.

1 Introduction

Little is known about the specific kinds of questions programmers ask when evolving a code base and how well existing and proposed tools support those questions. Some previous work has focused on developing models of program comprehension, which are descriptions of the cognitive processes a programmer uses to build an understanding of a software system (e.g., [50], [34]). Other work has focused on how programmers perform change tasks, including how programmers use tools in that context (e.g., [13], [54]). These previous efforts do not consider in detail what a programmer needs to know about a code base when performing a change task, how the programmer finds that information, nor how well tools support those activities.

To address this gap, we undertook two qualitative studies. In each of these studies, we observed programmers making source changes to medium (20 KLOC) to large-sized (over 1 million LOC) code bases. To structure our data collection and the analysis of our data, we used a *grounded theory* approach [16], [63]. Based on our analysis of the data from these user studies, as well as an analysis of the support that current programming tools provide for these activities, this research makes three key contributions. The first contribution is a catalog of 44 types of questions programmers ask, organized into four categories based on the kind and scope of information needed to answer a question. The second contribution is a description of the behavior we observed around answering those questions. The third contribution is a description of how well tools support a programmer in answering questions. Based on these results, we discuss the support that is missing from existing programming tools.

Section 2 of this paper compares the work presented in this paper to previous efforts in the area of program comprehension and empirical studies of how programmers manage change tasks. Section 3 describes the two studies we performed. Section 4 presents the 44 types of questions organized around four top-level categories and a description of the behavior we observed around answering questions. Section 5 considers the support existing research and industry tools provide for those activities. In Section 6, we discuss gaps in tool support. In Section 7, we discuss the limits of our results. We conclude with a summary in Section 8.

2 Related Work

In this section, we discuss three categories of related work. The first is the area of program comprehension, in particular efforts to use theories about program comprehension to inform tool design (see Section 2.1). The second covers work involving the analysis of programmers' questions (see Section 2.2). The third category includes empirical studies that have looked at how programmers use tools and generally how they carry out change tasks and other programming activities (see Section 2.3). Our review of these studies includes a discussion of studies that use similar research methods. We leave a discussion of the tool support available for various programming activities to

- *J. Sillito is with the Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, AB, T2N 1N4 Canada. E-mail: sillito@ucalgary.ca.*
- *G.C. Murphy and K. De Volder are with the Department of Computer Science, University of British Columbia, ICICS/CS Building, 201-2366 Main Mall, Vancouver, BC, V6T 1Z4 Canada. E-mail: {murphy, kdvolder}@cs.ubc.ca.*

Manuscript received 11 May 2007; revised 24 Nov. 2007; accepted 27 Mar. 2008; published online 21 Apr. 2008.
Recommended for acceptance by M. Young and P. Devanbu.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2007-05-0161.
Digital Object Identifier no. 10.1109/TSE.2008.26.

Section 5, in which we analyze the support a wide range of research and industry tools provide for answering the questions we found that programmers ask.

2.1 Program Comprehension

Work in the area of program comprehension has focused on proposing cognitive models that describe the cognitive processes and information structures programmers use to form a *mental model*, which is a programmer's mental representation of the program being maintained.

Theories by Brooks [5], Koenemann and Robertson [34], and Soloway and Ehrlich [56] suggest program comprehension occurs in a predominantly *top-down* manner. Brooks' model, for example, proposes that programmers comprehend a system by reconstructing knowledge about the domain of the program and by working to map that knowledge to the source code. According to the *bottom-up* theory, programmers first read individual statements in the code and then mentally group those statements into higher-level abstractions (capturing control-flow or data-flow, for example). These abstractions are in turn aggregated until this recursive process produces a sufficiently high-level understanding of the program. Two theories that propose a bottom-up approach are Pennington's model [45] and Shneiderman and Mayer's cognitive framework [51].

Other researchers have proposed theories suggesting that programmers combine strategies. For example, Littman et al. noted that programmers use either a *systematic* strategy or an *as-needed strategy* [38]. Letovsky's *knowledge-based* understanding model proposes that programmers work "opportunistically," using both bottom-up and top-down strategies [36], [35]. Von Mayrhauser and Vans propose a model that they call the *integrated metamodel* that combines four components: the top-down model, the program model, the situation model, and a knowledge base [68].

In contrast, our work has not focused on proposing new models of program comprehension nor have we attempted to validate any of these existing models. Instead, we aim to complement this work by filling in important details that are often abstracted away by theories of comprehension. To accomplish this goal, we have thoroughly analyzed empirical information collected as programmers performed software evolution tasks. We have also analyzed programmers' behavior around discovering that information along with the role that tools play in the answering process.

Work in the area of program comprehension has also striven to inform the design of programming tools to support various program understanding processes. Von Mayrhauser and Vans, for instance, have documented the tasks and subtasks programmers need to perform as part of understanding a system and have developed an associated list of *information needs* and *tool capabilities* to support those information needs. Similarly, based on a number of program comprehension models, Storey et al. present a hierarchy of cognitive issues or design elements to be considered during the design of a software exploration or visualization tool [60].

Walenstein takes a different approach to bridging the gap between cognitive models and tool design [70], [69]. His goal is a more solid theoretical grounding for the design of programming tools based on program comprehension theories, as well as other cognitive theories. Walenstein's *theory of cognitive support* describes the mental assistance tools can provide. Walenstein claims that such a theory can be used to rationalize tool features in terms of their support for cognition. Example principles on which support might be based include redistribution (moving cognitive resources to external artifacts) and perceptual substitution (transforming a task into a variant that is cognitively easier).

Our work takes a different approach to influencing the design of tools. Rather than beginning with models of cognition or other cognitive theories, we begin with observations about how programmers manage a change task and from those observations we build an understanding of the associated activities. In particular, we aim to use qualitative studies to fill in details around the specific questions programmers ask and how they use tools to answer those questions. We believe these details provide an important connection between program comprehension theories and programming tool research and design.

2.2 Analysis of Programmer's Questions

The work most similar to ours has investigated the questions programmers ask or the information they need to perform their work. Johnson and Erdem extracted and analyzed questions posted to Usenet newsgroups [29]. These questions were classified as goal oriented (requested help to achieve task-specific goals), symptom oriented (why something is going wrong), and system oriented (requested information for identifying system objects or functions). Herbsleb and Kuwana have empirically studied questions asked by software designers during real design meetings in three organizations [23]. They determined the types of questions asked, as well as how frequently they were asked. Our work is similar but targets a different part of the development process, namely, the question programmers ask while performing a change task to a system.

Letovsky presents observations of programmer activities, which he calls *inquiries*, and documents five kinds of conjectures programmers make (why, how, what, whether, and discrepancy) [36], [37]. Erdos and Sneed suggest, based on their personal experience, that seven questions need to be answered for a programmer to maintain a program that is only partially understood. These questions include *where is a particular subroutine/procedure invoked* and *what are the arguments and results of a given function* [12]. Building on the Usenet study by Johnson and Erdem [29], Erdem et al. have developed a model of the questions that programmers ask [11]. In their model, a question is represented based on its topic, the question type, and the relation type. We contribute to this body of knowledge by developing a more comprehensive list of questions, including questions at a higher level than those captured in this work.

Ko et al. conducted a study about the information needs of collocated software teams [31]. They identified 21 questions asked by programmers, in seven categories:

1. writing code,
2. submitting a change,
3. triaging bugs,

4. reproducing a failure,
5. understanding execution behavior,
6. reasoning about design, and
7. maintaining awareness.

Questions in the first category are about how to use and coordinate particular functions and data structures and map naturally to questions our participants asked. Questions in the second, third, and seventh categories deal with team issues or activities not considered in our study. The five questions in the fourth and fifth categories, such as *In what situations does this failure occur?*, relate to a large number of more specific questions identified in our work. Questions in category six (reasoning about design) were less prevalent in our studies. The study by Ko et al. focused on a wider range of activities and a broader context than our studies, resulting in a different range of questions. Our results provide a more detailed picture of what programmers need to understand specifically as they perform a change task.

2.3 Empirical Studies of Change Tasks

Other work has studied the situation of programmers performing change tasks from a number of different perspectives. Many of these efforts have explored the use of programming tools. For example, Storey et al. carried out a user study focused on how program understanding tools enhance or change the way that programmers understand programs [62]. In their study, 30 participants used various research tools to solve program understanding tasks on a small system. Based on their results, Storey et al. suggest that tools should support multiple strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration. In contrast to our work, Storey et al.'s work did not attempt to analyze specifically what programmers need to understand.

More similar to our studies are efforts that qualitatively examine the work practices of programmers. For example, Flor and Hutchins used distributed cognition to study a single pair of programmers performing a straightforward change task [13]. We use a similar study setup but with a larger participant pool and a more involved set of change tasks, with the goal of more broadly understanding the challenges programmers encounter. As another example, Singer et al. studied the daily activities of software engineers [54]. We focus more closely on the activities directly involved in performing a change task, producing a complementary study at a finer scale of analysis.

Four recent studies have focused on the use of current development environments (as do our studies). Robillard et al. characterize how programmers who are successful at maintenance tasks typically navigate a code base [48]. Deline et al. report on a formative observational study also focusing on navigation [9]. Our study differs from these in more broadly considering the process of asking and answering questions, rather than focusing exclusively on navigation. Ko et al. report on a study in which Java programmers used the Eclipse[1] development environment to work on five maintenance tasks on a small program [33]. Their intent was to gather design requirements for a maintenance-oriented development environment. Our study differs in focusing on a more realistic situation involving larger code bases and more involved tasks. Our analysis differs in that we aim specifically to understand what questions programmers ask and how they answer those questions. De Alwis and Murphy report on a field study about how programmers experience disorientation when using the Eclipse Java integrated development environment (IDE) [1]. They analyzed their data using the theory of visual momentum [74], identifying three factors that may lead to disorientation: the absence of connecting navigation context during program exploration, thrashing between displays to view necessary pieces of code, and the pursuit of sometimes unrelated subtasks. In contrast, our analysis has not employed the theory of visual momentum and has focused on questions and answers rather than disorientation.

1. http://www.eclipse.org, verified March 2008.

3 Research Approach

To investigate the detailed questions that arise during the programming activity associated with a software evolution task, we undertook two studies [53], [52]. Study one was conducted in a laboratory setting with nine participants working on a code base that was new to them. The goal of this study was to observe programmers performing significant change tasks using state-of-the-practice development tools. Specifically, these tasks were selected to require participants to make changes to several different source files. Study two was conducted in an industrial setting with 16 participants working on a code base for which they had responsibility. These two studies have allowed us to observe programmers in situations that vary along several dimensions, including the programming tools used, the type of change task, the system, paired versus individual programming, and the level of prior knowledge of the code base. The range of differences between sessions and between studies limits our ability to directly compare the differences and similarities along particular dimensions; however, the differences have increased our ability to generate an extensive set of questions programmers ask. Details of each study are presented in Sections 3.1 and 3.2.

To structure our data collection and the analysis of our data, we used a grounded theory approach, which is an emergent process intended to support the production of a theory that "fits" or "works" to explain a situation of interest [16], [63]. Grounded theory analysis revolves around various coding procedures that aim to identify, develop, and relate the concepts. As categories emerge, further selective sampling can be performed to gather more information, often with a focus on exploring variation within those categories. The aim here is to build rather than test theory and the specific result of this process is a theoretical understanding of the situation of interest that is grounded in the data collected.

3.1 Study One: Laboratory-Based Investigation

We designed study one to be as realistic as possible. Specifically, we used participants with development experience, real change tasks, and a nontrivial code base. We refer to

TABLE 1
Session Number, Driver, Observer, and Assigned Task for Each Session in Study One

Session	Driver	Observer	Task
1.1	N7	N3	484
1.2	N4	N1	1622
1.3	N4	N7	1021
1.4	N6	N4	1622
1.5	N5	N3	1622
1.6	N5	N2	1021
1.7	N3	N1	1622
1.8	N7	N5	2718
1.9	N8	N6	2718
1.10	N8	N2	1622a
1.11	N9	N2	1622a
1.12	N9	N6	1622a

TABLE 2
Tasks from Study One Along with an Estimate of the Number of Files Needed to be Changed to Perform The Task

Task	Files	Description
484	6	Make the font size for the interface configurable from the GUI.
1021	4	Add drag and drop support for changing association ends.
1622	9	Add property panel support for change, time and signal event types.
1622a	9	Add textual annotation support for change, time and signal event types. (A variation of task 1622.)
2718	13	Fix a model saving error that occurs after a use case with extends relationships is deleted from the model.

Numbers refer to IDs in the ArgoUML issue tracking system.

the nine participants (N1...N9) as newcomers as they were working on code that was new to them. All nine participants are male and were computer science graduate students with varying amounts of previous development experience, including experience with the Java programming language. Participants N1, N2, and N3 had five or more years of professional development experience. Participants N4, N5, and N6 had between two and five years of professional development experience. Participants N7, N8, and N9 had no professional development experience but did have one or more years of programming experience in the context of academic research projects. All participants had at least one year of experience using Eclipse for Java development. Each participant participated in two or three sessions (see Table 1).

The study involved 12 sessions (1.1...1.12). In each session, two participants performed an assigned task as a pair working side by side at one computer. We chose to study pairs of programmers because we believed that the discussion between the pair as they worked on the change task would allow us to learn what information they were looking for and why particular actions were being taken during the task, similar to earlier efforts (e.g., [13] and [41]). Following the terminology of Williams et al. [73], we use the term "driver" for the participant assigned to control the mouse and keyboard and "observer" for the participant working with the driver. In most sessions, the least experienced programmer was asked to be the driver. This choice was intended to encourage the more experienced programmer to be explicit about their intentions. None of the participants had significant previous experience working in pairs. The pairings are summarized in Table 1.

In each session, the programming pair was given 45 minutes to work on a change task using the Eclipse Java development environment (version 3.0.1), a widely used IDE that we consider representative of the state-of-the-practice. Participants were stopped after the 45 minutes elapsed, regardless of how much progress had been made. No effort was made to quantify how much of the task had been completed. The experimenter (the first author of this paper), who was present during each session, then briefly interviewed the participants about their experience. The interviews were informal and focused on the challenges faced by the pair, their strategy, how they felt about their progress, and what they would expect to do if they were continuing with the task. During each session, an audio recording was made of the discussion between the pair of participants, a video of the screen was captured, and a log of various Eclipse navigation and section events was made.

Table 2 describes the change tasks assigned to participants. The tasks were all enhancements or bug fixes to the ArgoUML[2] code base (versions 0.9, 0.13, and 0.16). ArgoUML is an open source UML modeling tool implemented in Java. It is comprised of roughly 60 KLOC. The tasks were complex, completed tasks chosen from ArgoUML's issue-tracking system. Table 2 also gives an estimate of the number of files that would need to be modified to successfully perform the change task. This number is based on the revision history from the ArgoUML project and should only be considered approximate because there are likely to be multiple ways to complete a change and multiple smaller changes are sometimes committed at the same time. However, these estimates illustrate that these tasks were based on complex nonlocal changes. During the study, participants worked with a version of the code base that predated the task completion by the ArgoUML team.

The ArgoUML code base is not extensively documented; however, there is some API documentation (in Javadoc[3] format) available for the project source code, as well as for several dependent libraries. During the study, the documentation for the most task relevant libraries was made available to our participants.

We did not expect that the participants would be able to complete the task in the time allotted, but we believed they would be able to make significant progress. Participants were asked to accomplish as much as possible on the given task but to not be concerned if they could not complete the task. In four of the sessions, 1.4, 1.7, 1.11, and 1.12, the pair of participants were asked to continue working on a task that one of them had commenced in a previous session, allowing us to gather data about later stages of work on a task. For example, session 1.4 involved participants N6 and N4 continuing on with the work that participant N6 began in session 1.2. Table 1 shows the assignment of tasks to sessions.

2. http://argouml.tigris.org, verified March 2008.
3. http://java.sun.com/j2se/javadoc/.

TABLE 3
Session Number, Participant(s), the Approximate Amount of Experience a Participant Had with the Code Base, the Programming Language(s) for the Target System, and the Primary Tools for Each Session of Study Two

Session	Part.	Experience	Languages	Tools
2.1	E1	8 years	C++, Tcl	Emacs, DDD, Virtual desktops
2.2	E2	8 years	C++, Tcl	Emacs (split window)
2.3	E3	4 years	C#, XSLT	Visual Studio, Biztalk Orchestration
2.4	E4	2.5 years	C#	Visual Studio, Biztalk Orchestration
2.5	E5	5 years	C#, ASP	Visual Studio
2.6	E6, E7	6 months	C#, ASP	Visual Studio, NetMeeting
2.7	E8	4 months	Java	Netbeans
2.8	E9	4 years	SQL, MDX	Visual Studio, Enterprise Manager, Query Analyzer, Analysis Manager
2.9	E10	4 months	C++, Batch	Notepad, Visual Studio
2.10	E11	4 years	C (embedded)	Proprietary loading and debugging tools
2.11	E12	6 months	C, C++	Visual Studio (two instances)
2.12	E13	7 months	HTML	UltraEdit, Proprietary Document Manager
2.13	E14	2 years	C	Emacs (split window)
2.14	E15	6 months	XML, Java	VIM (two instances)
2.15	E16	3 years	C	VI (two instances), GDB

3.2 Study Two: Industry-Based Investigation

Study two involved 16 programmers (E1...E16) in an industrial setting. We observed individual programmers, rather than pairs, because that was the normal work situation of the participants. Each of the sessions (numbered 2.1 to 2.15) is summarized in Table 3. One session from this study was exceptional in that two participants (E6 and E7) worked together because that was how the pair were accustomed to working. Three of the 16 participants (E9, E11, and E16) are female. All participants were professional programmers employed by the same large technology company. Several of the participants worked in the same groups within that company (E1 and E2; E6 and E7; E3, E4, and E5; E14 and E15) and, as a result, worked on similar code bases. In this study, the participants worked with code on which they had experience, though the amount of experience varied significantly (from a just few months to eight years). In addition to project source code, our participants also had access to all of the project documentation they would normally use.

Participants were observed as they worked on a change task to a software system for which they had responsibility. The systems were implemented in a range of languages and the participants used the tools that they would normally use. For example, participant E1 worked on a C++ [64] and Tcl [14] code base using tools such as Emacs [57] and DDD (a front end to GDB[4]), while E3 worked on a C# [21] and XSLT [66] code base using Microsoft Visual Studio.[5]

The tasks were selected by the participants. Each participant was asked, in advance of the session, to select a task that would be "involved, not a simple local fix." In each session, the programmer was asked to describe the task he or she had selected and then to spend about 30 minutes working on that task. Each participant was asked to think aloud while working on the task [67]. No think-aloud training or practice period was provided for participants. After each session, the experimenter (the first author of this paper) interviewed the participant (or participants) about their experience. The interviews were informal and focused on the challenges faced and their use of tools. An audio recording and field notes were made during each session, including during the interview portion.

4. http://sourceware.org/gdb/documentation/, verified March 2008.
5. http://msdn2.microsoft.com/en-us/vstudio/, verified March 2008.

4 Questions in Context

Our analysis focused on discovering the questions programmers had asked about the system and exploring similarities, connections, and differences among those questions. The analysis began with the first author of the paper coding the audio data to produce a list of specific questions asked by our participants. We only included questions targeting the code base in our analysis, which meant that some questions were excluded. For example, questions about how to approach the task and how a particular tool worked were not included. As a research group, we discussed the list of identified questions, finding that many of the questions asked were roughly the same, except for minor situational differences. We then developed generic versions of the questions that slightly abstract from the specifics of a particular situation and code base. For example, N4 asked the question "*how does [MAssociation] relate to [FigAssociation]?*" which can be stated more generically as *how are these types or objects related?* (see question 22).

Taking this further, we compared the generic questions and found that many of the similarities and differences could be understood in terms of the amount and type of information required to answer a given question. This observation became the foundation for our categorization

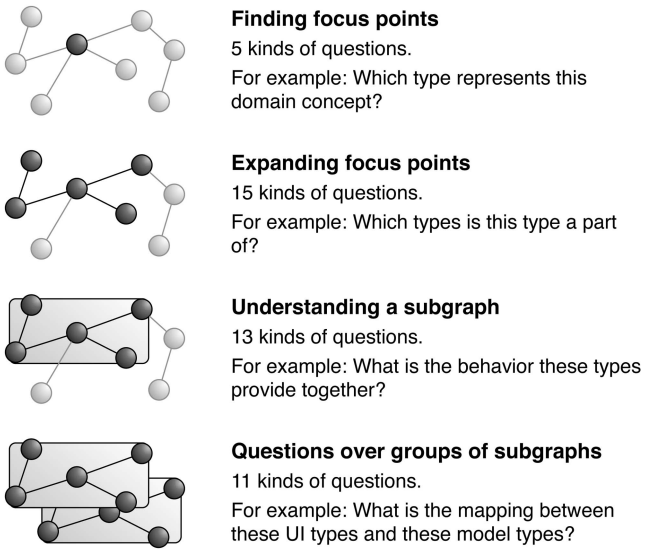

Fig. 1. An overview of the four categories of questions asked by our participants. Each is illustrated by a diagram depicting source code entities along with connections between those entities.

of the questions. If we consider a code base as a graph of entities (methods and fields, for example) and relationships between those entities (references and calls, for example), answering any given question requires considering some subgraph of the code base. The properties of the subgraph are the basis for our categorization, as illustrated in Fig. 1. Questions in the first category are about discovering a focus point in the graph. Questions in the second category are about a given entity and other entities directly related to it (i.e., questions that build on an entity). Questions in the third category are about understanding a number of entities and relationships together (i.e., questions about understanding a subgraph). Questions in the final category are over such connected groups: how they relate to each other or to the rest of the system (i.e., questions over groups of subgraphs).

Although other categorizations of these questions are possible, we present this categorization because it highlights the types and scope of information needed to answer the given questions, which is relevant to the design of supporting tools. This categorization also captures some intuitive sense of the various levels of questions asked and it helps explain various kinds of relationships between questions such as a question-subquestion relationship.

The levels suggested by our categories bear some resemblance to the hierarchy of information implied by various cognitive models (see Section 2.1). However, the order in which we present the categories is not representative of how the questions were necessarily asked and it is not our intention to suggest that our data supports a particular model such as the bottom-up model (e.g., [50]). However, at times, questions in the first category (finding focus points) were precursors to questions in the second category (expanding focus points), though these activities were not necessarily at the beginning of the session. Generally, we observed that participants often jumped around between various activities or explorations, at times leaving questions only partially answered (*"figure that out later"* [E8]), sometimes forgetting what they had learned (*"did we look at MAssociation? What was that?"* [N4]), sometimes abandoning an exploration path and beginning again (*"I guess we're on the wrong track there. Where is the earliest place we know that we can set a break point?"* [N2]), and sometimes returning to previous questions (*"I am still kind of curious..."* [N1]).

The four Sections (4.1, 4.2, 4.3, and 4.4) present the 44 different types of questions our participants asked, organized by category. These sections also present observations about how our participants used programming tools in answering their questions.

4.1 Finding Focus Points

One category of questions asked by our participants, the newcomers from the first study in particular, focused on finding points in the code that were relevant to the task. The participants in the first study naturally began a session knowing little or nothing about the code and, often, they were interested in finding any *"starting point"* [N9]. For example, N3 and N5 began session 1.5 by discussing *"where do we start?"* [N5]. Such questions were asked at the beginning of sessions but also as participants began to explore a new part of the system or generally needed a new focus point. These are perhaps similar to what Wilde and Casey call "places to start looking" [72].

These questions were at times about finding methods or types that correspond to domain concepts: *"I want to try and find the extends relationship [i.e., a concept from the domain of UML editors]"* [N5] and *"my idea is to see if we can find a representation of this transition"* [N1]. In other times, there were questions about finding code corresponding to UI elements or the text in an error message: *"what object refers to this actual [UI text]?"* [N7] and *"do a search for spline"* [N3] (where spline was the text in a tool tip). We observed five types of questions asked in this category. Each question is followed by a list of the sessions in which we observed it being asked:

1. Which type represents this domain concept or this UI element or action? (1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9)
2. Where in the code is the text in this error message or UI element? (1.1, 1.5, 1.9)
3. Where is there any code involved in the implementation of this behavior? (1.1, 1.2, 1.3, 1.5, 1.6, 1.10, 1.11, 2.11, 2.13)
4. Is there a precedent or exemplar for this? (1.1, 1.10, 1.12, 2.4, 2.6, 2.14, 2.15)
5. Is there an entity named something like this in that unit (project, package, or class, say)? (1.1, 1.2, 1.4, 1.5, 1.6, 1.10, 2.9)

To answer these questions, our participants often used text-based searches or Eclipse's Open Type Tool, which allows programmers to find types (classes or interfaces) by specifying a name or part of a name. Other questions, like question 5, were less amenable to text-based searches because the participants often had only a general idea of the sort of name for which they were looking. Instead, scrolling/scanning through code or overviews was used. At times, the number of search results or candidates

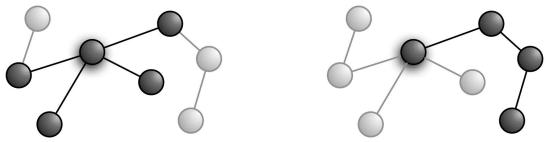

Fig. 2. A depiction of two observed patterns of questions: (a) multiple questions about the same entity and (b) a series of questions where each subsequent question is about a newly discovered entity.

otherwise identified was quite large and a fundamental question that needed to be answered was *what is relevant?*

In several sessions, the debugger was used to help answer questions of relevancy. Participants set break points in candidate locations (without necessarily first looking closely at the code) and ran the application to see which, if any, of those break points were encountered during the execution of a given feature. If none were encountered, this process was repeated with new candidate points. N6 explained his use of the debugger: *"I thought maybe these classes are not even relevant, even though they look like they should be. So I get confidence in my hypothesis, just that I am on the right track"* [N6].

4.2 Expanding Focus Points

A second category of questions was about expanding a given entity believed to be related to the task, often by exploring relationships. For example, after finding a method relevant to the task, N3 asked the following sequence of questions: *"what class is this [in]?"*; *"what does it inherit from?"*; *"now where are these NavPerspective's [a type] used?"*; and then *"what [container] are they put into?"*. With these kinds of questions, the participants aimed to learn more about a given entity and to find more information relevant to the task.

Sometimes we observed a series of questions about the same entity, forming a star pattern, as depicted in Fig. 2a (showing source code entities and connections between entities). At other times, we observed a series of questions where each subsequent question started from an entity discovered as an answer to a previous question, forming a linear pattern, as depicted in Fig. 2b.

Some questions in this category were questions about types, including questions about the static structure of types: *"are there any sibling classes?"* [N3] or *"what is the type of this object?"* [E16]. We observed six such questions:

6. What are the parts of this type? (1.2, 1.5, 1.6, 1.7, 1.8, 1.10, 1.11, 2.15)
7. Which types is this type a part of? (1.2, 1.5)
8. Where does this type fit in the type hierarchy? (1.1, 1.2, 1.3, 1.5, 1.6, 1.12)
9. Does this type have any siblings in the type hierarchy? (1.5, 1.11)
10. Where is this field declared in the type hierarchy? (1.5, 1.7)
11. Who implements this interface or these abstract methods? (1.2, 1.5, 1.6, 1.7, 1.10)

Other questions in this category focused on discovering entities and relationships that capture incoming connections to a given entity, such as *"let's see who sends this"* [N1]; *"so where does that method get called, can you look for references?"* [N2]; *"who is using the factory?"* [N4]; and *"now I look to see where this gets set"* [E15]:

12. Where is this method called or type referenced? (1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.10, 1.11, 1.12, 2.1, 2.4)
13. When during the execution is this method called? (1.2, 1.4, 1.5, 2.15)
14. Where are instances of this class created? (1.2, 1.3, 1.5, 1.7, 1.8, 1.10)
15. Where is this variable or data structure being accessed? (1.4, 1.5, 1.6, 1.7, 1.12, 2.1, 2.8, 2.14)
16. What data can we access from this object? (1.8, 2.15)

Questions 12 and 13 are similar in that both are about a call to a particular method. The distinction is that, with question 12, the participant is asking with respect to the static structure, whereas, with question 13, the participant is asking with respect to a particular execution of the system.

Finally, there were also questions around outgoing connections from a given entity, many of which were aimed at learning about the behavior of that entity, including questions about callees and argument types (*"I wonder what [this argument] is?"* [N1]):

17. What does the declaration or definition of this look like? (1.2, 1.5, 1.8, 1.10, 1.11, 2.1, 2.2, 2.11, 2.13, 2.15)
18. What are the arguments to this function? (1.3, 1.4, 1.5, 1.7, 1.8, 1.10, 1.11, 1.12)
19. What are the values of these arguments at runtime? (1.4, 1.9, 1.12, 2.15)
20. What data is being modified in this code? (1.6, 1.11)

Many questions in this category could be answered directly with the tools available. For example, the question *"how it is that I reach it"*[N6] (see question 13) was answered using the call stack viewer in the debugger. Others could only be approximated with the available tools. For example, the question *"what classes have MEvents as fields?"* [N3] (see question 7) could be approximated by a references search. In cases like these and also for questions about connections involving polymorphism, inheritance events, and reflection (*"they are making it so convoluted, with all the reflection"*[N6]), the results were more noisy and more difficult to interpret. In some cases, participants were able to switch tools or otherwise refine their use of tools to get a more precise answer. For example, *"maybe I can filter this a bit more, so we get less records"* [E9].

4.3 Understanding a Subgraph

A third category of questions was about building an understanding of concepts in the code that involved multiple relationships and entities. Answering these questions required the right details, as well as an understanding of the overall structure of the relevant subgraph: *"we really have to get a good understanding of the whole"* [N3]. This need is expressed in a comment by participant N6 that exposes a desire to understand the results of several searches together: *"I was starting to forget who was calling what, especially because there is only one search panel at a time that I can see"* [N6].

To see the distinction between this category and the one just described in Section 4.2, consider questions 6 (*What are the parts of this type?*) and 7 (*Which types is this type a part of?*) from the previous category and question 22 (*How are these types or objects related?*) included as part of the category described in this section. Questions 6 and 7 are about direct relationships to a particular source code entity, while question 22 is similar but requires considering a subgraph of the system together.

Some questions in this category were aimed at understanding certain behavior (*"we could trace through how it does it's work"* [N1]) and the structure of specific parts of the code base (*"I thought it would tell me something about the structure of the model"* [N6]). Some of the questions around these issues aimed at understanding "why" things were the way they were and what the logic was behind a given decomposition (*"why they're doing that"* [E14]).

21. How are instances of these types created and assembled? (1.1, 1.2, 1.4, 1.7, 1.9, 1.10, 1.11, 1.12)
22. How are these types or objects related? (whole-part) (1.2, 1.10)
23. How is this feature or concern (object ownership, UI control, etc.) implemented? (1.1, 1.2, 1.4, 1.7, 1.11, 1.12, 2.1, 2.13)
24. What in this structure distinguishes these cases? (1.2, 1.12, 2.8)
25. What is the behavior that these types provide together and how is it distributed over the types? (1.1, 1.2, 1.3, 1.4, 1.6, 1.11, 2.11)
26. What is the "correct" way to use or access this data structure? (1.8, 2.3, 2.15)
27. How does this data structure look at runtime? (1.8, 1.9, 1.10, 2.15)

Other questions in this category were about data and control flow. Note that these are not questions such as *what calls this method* but instead were about the flow of control or data involving multiple calls and entities such as *"how do I get this value to here?"* [E15].

28. How can data be passed to (or accessed at) this point in the code? (1.5, 1.6, 1.8, 1.12, 2.14)
29. How is control getting (from here to) here? (1.3, 1.4)
30. Why is not control reaching this point in the code? (1.4, 1.9, 1.10, 1.12, 2.1, 2.10)
31. Which execution path is being taken in this case? (1.2, 1.3, 1.7, 1.9, 1.12, 2.2, 2.9)
32. Under what circumstances is this method called or exception thrown? (1.3, 1.4, 1.5, 1.9)
33. What parts of this data structure are accessed in this code? (1.6, 1.8, 1.12)

At times, an answer to a question in this category was pursued by asking a number of other lower-level questions. For example, answering question 14 (*Where are instances of this class created?*) may provide information relevant to answering question 21 (*How are instances of these types created and assembled?*). We observed that, during this process, participants often revisited entities, repeating questions such as *"I forgot what we figured out from that"* [N3] and *"I want to have another look at this guy"* [N8]. Using lower level questions in this way meant that higher level questions were answered in pieces (perhaps using a number of different tools) and we observed that a participant seeing or discovering relevant entities and relationships individually was not always sufficient to mentally build an answer to the questions in this category; one participant noted *"it gets very hard to think in your head how that works"* [E14]. For example, losing track of the temporal ordering of method calls and of structural relationships that they had already investigated was a source of confusion for the participants in session 1.10: *"why is the name already set?"* [N2] and *"why is the namespace null?"* [N2].

4.4 Questions over Groups of Subgraphs

The fourth category of questions we observed in our studies includes questions over related groups of subgraphs. The questions already described in Section 4.3 involved understanding a subgraph, while the questions in this category involve understanding the relationships between multiple subgraphs or understanding the interaction between a subgraph and the rest of the system. For example, question 29 (*How is control getting (from here to) here?*) in the previous category is about understanding a particular flow through a number of methods, whereas question 34 presented in this category is about how two related control-flows vary.

Questions around comparing or contrasting groups of subgraphs included questions such as *"what do these things have that are different than each other?"* [N1] and *"I am jumping between the source and the header trying to compare what was moved out"* [E2]. For example, participant N6 was interested in learning about differences between four different types: *"I looked at what was different between those four classes, and at first, I tried looking at the implementation [i.e., the source code for the classes], but I thought, what might be more interesting is to see the call event called from some place the other ones are not"* [N6].

Several participants in the second study used split Emacs windows (E2 and E14), multiple monitors (E12), or multiple windows (E16), which seemed to help with answering these questions around making comparisons: *"so I can look at both files, edit both of them without having to click from window to window"* [E14] and *"using two monitors I can look at this source code, as well as the engine code itself without having to swap windows"* [E12]. With these arrangements, more (though not all) of the information that they were comparing could be seen side by side. We also observed questions about how two subgraphs were connected; for example, question 37 was asked after participants had discovered various user interface types and various model types and needed to understand the connection between these two groups.

34. How does the system behavior vary over these types or cases? (1.3, 1.4, 2.14)
35. What are the differences between these files or types? (1.2, 2.1, 2.2, 2.13, 2.15)
36. What is the difference between these similar parts of the code (e.g., between sets of methods)? (1.7, 1.8, 1.11, 2.6, 2.11, 2.14, 2.15)

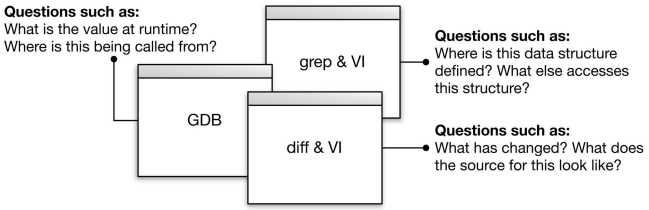

Fig. 3. Participant E16's arrangement of windows and associated tools. Also shown are the types of questions asked in each window.

37. What is the mapping between these UI types and these model types? (1.1, 1.2, 1.5, 1.7)

Given an understanding of a number of structures, our participants asked questions around how to change those structures (see questions 38 and 39, below). Specific examples include *"as long as we can figure out how to fit into the existing framework, we should be OK"* [N3] and *"how to sort of decouple it and add sort of another layer of choice?"* [N6].

They also asked questions around determining the impact of their (proposed) changes, including asking questions around understanding how the structures of interest were connected with the rest of the system: *"there's a lot of the interactions between the different modules that aren't exactly understood"* [E10] and *"I find it hard to see what are all the things that are acting on the bits of code we are looking at"* [N9]. One participant was guided in making changes by the question *"what's the minimal impact to the source code I [can] have?"* [E12]. Question 41 below was asked by participants trying to determine whether or not their changes were correct:

38. Where should this branch be inserted or how should this case be handled? (1.4, 1.5, 1.6, 1.8, 1.9, 2.11, 2.15)
39. Where in the UI should this functionality be added? (1.1, 1.5, 1.7, 2.1, 2.6)
40. To move this feature into this code, what else needs to be moved? (2.7, 2.13)
41. How can we know that this object has been created and initialized correctly? (1.10, 1.12)
42. What will be (or has been) the direct impact of this change? (1.5, 1.7, 1.8, 1.10, 1.11, 1.12, 2.1, 2.2, 2.4, 2.6, 2.7, 2.8, 2.12, 2.15)
43. What will the total impact of this change be? (2.1, 2.2, 2.3, 2.4, 2.5, 2.9, 2.10, 2.11)
44. Will this completely solve the problem or provide the enhancement? (1.1, 1.9, 1.11, 2.12, 2.14)

The process of answering questions in this category involved multiple supporting questions and multiple tools. Working in this way, while producing a significant volume of results, does not necessarily result in an answer to the original question posed. For example, participant E16 used several different tools (GDB, diff,[6] grep,[7] and VI[8]). These tools were in three different windows, arranged as in Fig. 3. She explained her arrangement in this way: *"I [have] one where I am running the program, one where I am actually looking at the code, and one where I am just searching for other things."* By the end of the session, participant E16 still had not been able to answer her higher level question and decided she needed to begin the process again, making different choices about what lower level questions to ask, and how to use the available tools to answer those questions.

6. http://www.gnu.org/software/diffutils/manual, verified March 2008.
7. http://www.gnu.org/software/grep/doc/, verified March 2008.
8. http://www.vim.org/, verified March 2008.

4.5 Question Frequencies

Our study design and data analysis focused on identifying the range of questions that programmers' ask about a code base. It is natural to also question how often various questions arise. We present frequency data of the questions we observed in Table 4. The figure shows the number of times distinct questions of each type were asked in each session, along with totals for each study. For example, concrete questions corresponding to question 19 (*What are the values of these arguments at runtime?*) were asked once in each of sessions 1.4, 1.12, and 2.15, and twice in session 1.9. If a specific question is asked repeatedly in a session, it is counted only once in Table 4.

Various trends are visible in this frequency data. For example, the data illustrates that questions in the first three categories occurred more frequently during the first study than the second. On the other hand, questions in the fourth category (questions 34 to 44) occurred more frequently during the second study than the first. One explanation for the differences is that participants in the first study were newcomers to the code on which they were working, while participants in the second study were working with code with which they had experience. Another contributing factor may be that working with a pair (as the participants in study one did) encourages a participant to articulate more of his or her questions.

Any observations drawn from this frequency data must be treated by case. The sessions from which the data is drawn are limited in duration and varied in important ways. These and other limitations are discussed further in Section 7.

5 Analysis of Tool Support for Answering Questions

Our two user studies provided some insights into how programmers use tools to support the process of answering the questions we observed. To build a more complete understanding of the state-of-the-art in tool support for answering the questions and of where tool support is lacking, we have analyzed the ability of a wider range of tools (both industry and research tools) and techniques to support a programmer in answering each of the questions.

Our analysis consisted of analyzing the literature on programming tools and techniques for exploring source code. For each question our participants asked, we tried to identify a tool with support for answering that question and we qualitatively evaluated the level of support provided. This discussion is not intended to be a comprehensive survey of applicable tools or techniques. Our goal has not been to identify all tools or techniques applicable to each question but rather to determine whether or not one exists to address each question. We rate the level of support provided by the best available tool we found as

TABLE 4
The Number of Times Distinct Questions of Each Type That Were Asked in Each Session

	Question types	Study 1 Sessions													Study 2 Sessions														
		1	2	3	4	5	6	7	8	9	10	11	12	#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	#
1	Which type represents this domain concept or this UI element or action?	1	1	1		1	1	1	1	1				8																
2	Where in the code is the text in this error message or UI element?	2				1				1				4																
3	Where is there any code involved in the implementation of this behavior?	4	1	1		1	1				1	1		10											1		1			2
4	Is there a precedent or exemplar for this?	2									1		1	4				1		1								1	1	4
5	Is there an entity named something like this in that unit (project, package or class, say)?	2	3		2	1	1				2			11									1							1
6	What are the parts of this type?		3			1	1	1	1		2	2		11															1	1
7	Which types is this type a part of?		1			1								2																
8	Where does this type fit in the type hierarchy?	2	4	1		1	1						1	10																
9	Does this type have any siblings in the type hierarchy?					1						1		2																
10	Where is this field declared in the type hierarchy?					1		1						2																
11	Who implements this interface or these abstract methods?		1			1	1	1			1			5																
12	Where is this method called or type referenced?	4	5	2	4	2	2	5	2		3	3	1	33	1			1												2
13	When during the execution is this method called?		1		2	1								4															1	1
14	Where are instances of this class created?		1	1		2		2	1		1			8																
15	Where is this variable or data structure being accessed?				1	1	2	3					1	8	1							1						1		3
16	What data can we access from this object?								1					1															1	1
17	What does the declaration or definition of this look like?		4			2			3		2	2		13	1	1									1		1		1	5
18	What are the arguments to this function?			1	1	1		1	2		1	1	2	10																
19	What are the values of these arguments at runtime?				1					2			1	4															1	1
20	What data is being modified in this code?						1					1		2																
21	How are instances of these types created and assembled?	1	1		1			1		1	2	1	1	9																
22	How are these types or objects related? (whole-part)		2								1			3																
23	How is this feature or concern (object ownership, UI control, etc.) implemented?	3	1		2			1				2	3	12	1												2			3
24	What in this structure distinguishes these cases?		1										1	2								1								1
25	What is the behavior that these types provide together and how is it distributed over the types?	1	2	1	1		1					1		7											1					1
26	What is the "correct" way to use or access this data structure?	1							2					3			1												2	3
27	How does this data structure look at runtime?								1	1	1			3															3	3
28	How can data be passed to (or accessed at) this point in the code?					1	1		1				1	4														1		1
29	How is control getting (from here to) here?			1	1									2																
30	Why isn't control reaching this point in the code?				2					1	1		2	6	1									1						2
31	Which execution path is being taken in this case?		1	3				1		1			1	7		1							1							2
32	Under what circumstances is this method called or exception thrown?			2	4	1				1				8																
33	What parts of this data structure are accessed in this code?						1		1				1	3																
34	How does the system behavior vary over these types or cases?			1	1									2														1		1
35	What are the differences between these files or types?		1											1	1	3											3		1	8
36	What is the difference between these similar parts of the code (e.g., between sets of methods)?							1	1			1		3						1					1			1	1	4
37	What is the mapping between these UI types and these model types?	1	1			1		1						4																
38	Where should this branch be inserted or how should this case be handled?				1	1	1		1	1				5											1				1	2
39	Where in the UI should this functionality be added?	2				1		1						4	1					1										2
40	To move this feature into this code what else needs to be moved?																				1						1			2
41	How can we know that this object has been created and initialized correctly?										1		1	2																
42	What will be (or has been) the direct impact of this change?					1		1	1		2	1	1	7	2	2		2		2	1		3			2			1	15
43	What will be the total impact of this change?														1	1	1	1	2				1	1	1					9
44	Will this completely solve the problem or provide the enhancement?	1								1		1		3												1		1		2

full or *partial*. For a tool to be considered as providing full support for a given question, it has to allow a programmer to directly answer the question; otherwise, we rate the level of support as partial. This discussion is organized around the four categories of questions. Tables 5, 6, 7, and 8 summarize the techniques and tools applicable to answering each question and the level of support provided by those techniques and tools.

5.1 Answering Questions Around Finding Focus Points

Questions 1 (*Which type represents this domain concept or this UI element or action?*), 2 (*Where in the code is the text in this error message or UI element?*), and 5 (*Is there an entity named something like this in that unit (project, package or class, say)?*) all require finding a name or some text in the source code. Simple lexical search tools such as grep are often helpful in answering these questions. For instance, a programmer who uses grep to answer question 1 may form a hypothesis about possible names for types representing a given concept, formulate an appropriate regular expression, and then perform a search. IDEs often provide more focused support such as Eclipse's Open Type tool, which uses static structural analysis to limit the search to type names.

Question 5 can also be supported by source code editors and various overview tools. These are particularly helpful in situations where a hypothesis about the name to locate is

TABLE 5
A Summary of the Techniques and Tools Applicable to Answering Each of the Questions from the First Category

1	**Which type represents this domain concept or this UI element or action?** Full support: Lexical or static analysis based search (e.g., Eclipse's Open Type tool)
2	**Where in the code is the text in this error message or UI element?** Full support: Lexical or static analysis based search (e.g., grep)
3	**Where is there any code involved in the implementation of this behavior?** Partial support: Search with debugging support or feature location techniques (e.g., RECON3 [25])
4	**Is there a precedent or exemplar for this?** Partial support: Lexical or static analysis based search or example finding tools (e.g., CodeFinder, assuming an appropriate repository this could be considered *full* support [22])
5	**Is there an entity named something like this in that unit (project, package or class, say)?** Full support: Lexical or static analysis based search (e.g., grep)

TABLE 6
A Summary of the Techniques and Tools Applicable to Answering Each of the Questions from the Second Category

6	**What are the parts of this type?** Partial support: Source code editors or structural overviews (e.g., Eclipse's Outline View)
7	**Which types is this type a part of?** Full support: Cross-referencing tools (e.g., Masterscope [65])
8	**Where does this type fit in the type hierarchy?** Full support: Type hierarchy tools (e.g., Eclipse's Type Hierarchy)
9	**Does this type have any siblings in the type hierarchy?** Full support: Static analysis based type hierarchy tools (e.g., Eclipse's Type Hierarchy)
10	**Where is this field declared in the type hierarchy?** Full support: Static analysis based type hierarchy tools (e.g., Eclipse's Type Hierarchy)
11	**Who implements this interface or these abstract methods?** Full support: Static analysis based type hierarchy tools (e.g., Eclipse's search for implementors feature)
12	**Where is this method called or type referenced?** Full support: Cross-referencing tools (e.g., CScope [58])
13	**When during the execution is this method called?** Full support: Debugging tools (e.g., GDB)
14	**Where are instances of this class created?** Full support: Cross-referencing tools (e.g., CScope [58])
15	**Where is this variable or data structure being accessed?** Full support: Cross-referencing tools or slicing techniques (e.g., [20])
16	**What data can we access from this object?** Partial support: Source code editors or structural overviews (e.g., Eclipse's Outline View)
17	**What does the declaration or definition of this look like?** Full support: Lexical based cross-referencing tools (e.g., ctags)
18	**What are the arguments to this function?** Full support: Lexical based cross-referencing tools (e.g., ctags)
19	**What are the values of these arguments at runtime?** Full support: Debugging tools (e.g., GDB)
20	**What data is being modified in this code?** Partial support: Source code editor, data-flow analysis techniques or slicing techniques (e.g., [20])

TABLE 7
A Summary of the Techniques and Tools Applicable to Answering Each of the Questions from the Third Category

21	**How are instances of these types created and assembled?** Partial support: Visualization or browsing tools (e.g., SHriMP [61])
22	**How are these types or objects related? (whole-part)** Partial support: Visualization or browsing tools (e.g., SHriMP [61])
23	**How is this feature or concern (object ownership, UI control, etc.) implemented?** Partial support: Feature location techniques (e.g., software reconnaissance [72])
24	**What in this structure distinguishes these cases?** Partial support: Visualization or browsing tools (e.g., FEAT [49])
25	**What is the behavior that these types provide together and how is it distributed over the types?** Partial support: Visualization or browsing tools (e.g., Relo [55])
26	**What is the "correct" way to use or access this data structure?** Partial support: Visualization or browsing tools (e.g., SHriMP [61])
27	**How does this data structure look at runtime?** Partial support: Debugging and data structure visualization tools tools (e.g., Amethyst [43])
28	**How can data be passed to (or accessed at) this point in the code?** Partial support: Runtime visualization tools (e.g., BALSA [6])
29	**How is control getting (from here to) here?** Partial support: Visualization or browsing tools (e.g., a call hierarchy browser)
30	**Why isn't control reaching this point in the code?** Partial support: Debugging and slicing techniques (e.g., Whyline [32])
31	**Which execution path is being taken in this case?** Partial support: Debugging and slicing techniques (e.g., GDB)
32	**Under what circumstances is this method called or exception thrown?** Partial support: Debugging and visualization tools (e.g., BALSA [6])
33	**What parts of this data structure are accessed in this code?** Partial support: Source code editors or browsing (e.g., Eclipse's Outline View)

difficult to form. As one example, this difficulty was encountered in session 1.6, where the participants were looking *"for a move or a set destination or something like that"* [N2]. These participants scrolled and scanned through the source code in an editor and then used Eclipse's Outline View to try to find a relevant entity.

Answering question 3 (*Where is there any code involved in the implementation of this behavior?*) is about finding any point in the code relevant to a particular portion of a task. Several of our participants first used lexical or structural search tools to generate candidate types or methods. The participants then used additional searchers or debugging techniques to check the hypothesis that a given candidate was part of the behavior of interest (*"get confidence in my hypothesis"* [N6]). This multistep process was not always straightforward and at times included several failed attempts. Software reconnaissance is a test-based technique that uses a comparison of traces of different test cases to help locate relevant code and in some cases could reduce the exploration effort involved in answering question 3. RECON2 and RECON3 are examples of tools providing support for this technique [25].

Answering question 4 (*Is there a precedent or exemplar for this?*) requires identifying points in a code base that provide information about how to write certain types of code in the context of that code base. Lexical search or structural cross-referencing tools that allow programmers to elicit relationships between source code entities can also help [65], [58]. For instance, in the second study, when E15 was looking for examples of the use of a particular API, he used grep to find candidate locations in the code. In session 1.10, the participants (N2 and N8) looked for an example using Eclipse's cross-reference search tools. To make use of that example, they first copied and pasted it (*"should we just copy the code and see what happens?"* [N8]) and then made changes as needed. One challenge we observed for programmers in finding exemplars was in formulating a query that sufficiently captures the situation. In particular, there are cases when searching for a reference to one type or method produces many irrelevant results. Several research tools aim to address this problem, including CodeFinder [22], which supports queries over an example-based programming environment, and Strathcona [24], which automatically creates queries based on structural context.

In summary, answering these questions generally involves performing searches based on a hypothesis of what identifiers or other text were used, possibly based on information from the domain or the user interface of the system. Generally speaking, answering questions in this category is well supported, with all questions having at least partial support. The challenges that do exist in answering these questions stem from difficulties in formulating queries or in the volume of information returned

TABLE 8
A Summary of the Techniques and Tools Applicable to Answering Each of the Questions from the Fourth Category

34	**How does the system behavior vary over these types or cases?** Partial support: Structural searches and debugging techniques (e.g., GDB)
35	**What are the differences between these files or types?** Partial support: Line-based comparison tools (e.g., diff)
36	**What is the difference between these similar parts of the code (e.g., between sets of methods)?** Partial support: Line-based comparison tools (e.g., diff)
37	**What is the mapping between these UI types and these model types?** Partial support: Conceptual module querying [2]
38	**Where should this branch be inserted or how should this case be handled?** Partial support: Visualization and browsing tools (e.g., SHriMP [61])
39	**Where in the UI should this functionality be added?** Partial support: Visualization and browsing tools (e.g., SHriMP [61])
40	**To move this feature into this code what else needs to be moved?** Partial support: Conceptual module querying [2]
41	**How can we know that this object has been created and initialized correctly?** Partial support: Visualization and browsing tools (e.g., SHriMP [61])
42	**What will be (or has been) the direct impact of this change?** Partial support: Testing and impact analysis techniques (e.g., Chianti [47])
43	**What will be the total impact of this change?** Partial support: Testing and impact analysis techniques (e.g., Unit tests)
44	**Will this completely solve the problem or provide the enhancement?** Partial support: Testing techniques [3]

by various tools, much of which is irrelevant to the intended question.

5.2 Answering Questions Around Expanding a Focus Point

Questions 6 (*What are the parts of this type?*) and 16 (*What data can we access from this object?*) both require considering methods and fields of a given type. Source code editors or structural overview tools can help. For participants in the first study, Eclipse's Outline View provided such an overview. The participant in session 2.15 used vi (and to some extent GDB) to support answering these questions. In our two studies, we observed some situations where getting at the essential structure to answer these questions was not straightforward due to the volume of information presented by the various views. We observed participants carefully reading source code to answer question 20 (*What data is being modified in this code?*), which requires considering a block of code and determining (some of) its effects. Baniassad and Murphy [2] and Jackson [26] demonstrate that data-flow analysis techniques can be used to produce a list of effects that may allow a programmer to answer such questions more directly, though likely some additional investigation would be required.

Questions 7 (*Which types is this type a part of?*), 12 (*Where is this method called or type referenced?*), 14 (*Where are instances of this class created?*), and 15 (*Where is this variable or data structure being accessed?*) consider a type, method, or variable and ask about connections to it. For the most part, these can be answered relatively directly by tools using static structural analysis. We have observed that such tools are less helpful when Java reflection or other indirection obscured the control flow. Also, we have observed that, in some situations, answering a question such as 15 may require tools to analyze the data flow. In these cases, a tool based on slicing [71], [20], [76] or chopping [27] techniques for identifying code that impacts the value of a variable may help with answering the questions.

Questions 17 (*What does the declaration or definition of this look like?*) and 18 (*What are the arguments to this function?*) also involve following relationships between entities and can be similarly supported by static analysis techniques. For example, IDEs such as Eclipse and Visual Studio used in our studies support navigating to the declaration of a given type or variable. We observed that this feature was frequently used by our participants. Several participants (E1, E2, and E14) in the second study used a tool called ctags,[9] which is based on lexical analysis, to support this kind of navigation.

Questions 8 (*Where does this type fit in the type hierarchy?*), 9 (*Does this type have any siblings in the type hierarchy?*), 10 (*Where is this field declared in the type hierarchy?*), and 11 (*Who implements this interface or these abstract methods?*) all consider aspects of the type hierarchy of an object-oriented system. Static structural analysis techniques can be used to elicit the necessary information and various tools exist for displaying it. For example, Eclipse provides a Type Hierarchy View, one of the more frequently used tools by participants in study one.

Questions 13 (*When during the execution is this method called?*) and 19 (*What are the values of these arguments at runtime?*) consider the dynamic properties of a system, generally in the context of a particular point in an execution, as opposed to a questions such as 15 (*Where is this variable or data structure being accessed?*) that asks generally for the possible callers of a method or referencers of a type. These more specific questions can be answered using debugging tools that provide the ability to set a breakpoint. When the breakpoint is reached during execution, the call stack (to answer question 13) and the values of variables (to answer question 19) can be inspected. For example, we observed participant E16 use GDB to answer several instances of both of these questions.

In summary, answering questions from category two generally involves considering information about different types of relationships between source code entities. In most cases, eliciting this information is relatively well supported by static analysis-based tools such as those that have been available for many years (e.g., see [18], [44]). For other cases, debugging tools, overview tools, or data-flow techniques provide some support. Where there were challenges, they stemmed from various forms of indirection or the volume of information presented by the various tools. Despite this, answering these questions is relatively well supported by today's tools, as summarized in Table 6.

9. http://ctags.sourceforge.net/ctags.html, verified March 2008.

5.3 Answering Questions Around Understanding a Subgraph

Question 23 (*How is this feature or concern (object ownership, UI control, etc.) implemented?*) requires identifying what methods or types are involved in the implementation of a given concept, as well as understanding the relationships between them. This question is directly the aim of feature location techniques such as software reconnaissance (see Section 5.1) and the dependency graph method [8], which is a systematic approach involving asking a series of lower level questions to produce information toward answering a higher level question. Eisenbarth and Koschke combines dynamic and static analysis information to identify computation units that contribute to a feature [10]. These techniques provide information that can support a programmer in building an understanding of how a particular feature is implemented.

Answering questions 30 (*Why is not control reaching this point in the code?*) and 31 (*Which execution path is being taken in this case?*) requires understanding aspects of the dynamic control flow or data flow in a particular context. Debugging tools, including capture and replay tools (e.g., [59]), provide helpful information though the exploration effort can still be significant. An interrogative debugging tool by Ko and Myers called Whyline aims to help programmers both ask and answer these kinds of questions using a program slicing technique [32].

We observed that considering dynamic information about data flow and about control flow can aid in answering questions 27 (*How does this data structure look at runtime?*), 28 (*How can data be passed to (or accessed at) this point in the code?*), and 32 (*Under what circumstances is this method called or exception thrown?*). Our participants made frequent use of the debugger (GDB or the Eclipse Debug perspective) in this effort. However, answering these questions was challenging and involved a number of investigations around lower level questions. There have been several efforts to support the visualization of data structures and control flow at runtime [46], including the Amethyst (later called Pascal Genie) debugging tool [43], and BALSA (later called Zeus) [6], [7]. The information provided by these tools is similar to the information now provided by IDE's debugging tools.

Answering questions 21 (*How are instances of these types created and assembled?*), 22 (*How are these types or objects related? (whole part)*), 24 (*What in this structure distinguishes these cases?*), 25 (*What is the behavior that these types provide together, and how is it distributed over the types?*), and 26 (*What is the "correct" way to use or access this data structure?*) requires understanding a range of static and dynamic information. As described previously, to answer these questions, our participants used a number of tools to answer several supporting questions. In the process of attempting to understand the subgraph, participants often revisited entities believed to be relevant. Various code browsing tools (e.g., Lemma [40], FEAT [49], and JQuery [28]) have been developed to make the navigation or revisiting aspect of this process more direct. These browsing tools, as well as various general visualization tools (e.g., SHriMP [61]), provide support for bringing information together and, as a result, may provide some minimal support for answering questions such as 22, 24, and 25. However, the process still revolves around lower level questions and no direct support is provided to identifying relevant information.

Questions 29 (*How is control getting (from here to) here?*), 33 (*What parts of this data structure are accessed in this code?*), and 37 (*What is the mapping between these UI types and these model types?*) all consider two different sets of entities or points in the code and ask about the connections between them. For example, question 29 is about understanding the control-flow between two methods. A tool such as the Call Hierarchy Viewer provided in Eclipse can be used to produce information toward answering these questions, but the branching factor is high and we observed that our participants rarely used this viewer beyond two or three calls. The Relo tool [55] mentioned above supports a feature called Autobrowsing that models a simple directed exploration activity between two or more selected entities, which in some simple situations may help answer these questions about understanding connections.

In summary, tool support for answering questions in this category is limited. We found that, often, to answer these lower level questions, possibly less refined versions of these questions (i.e., ones with better tools support) must be asked, resulting in noisier result sets and the need to mentally put together answers. It is possible that visualization tools may make integrating this information easier.

5.4 Answering Questions About Groups of Subgraphs

Questions 34 (*How does the system behavior vary over these types or cases?*), 35 (*What are the differences between these files or types?*), and 36 (*What is the difference between these similar parts of the code (e.g., between sets of methods)?*) are about making comparisons between behavior, types, or methods. Generally, making comparisons is difficult, especially comparing behavior as needed for answering question 34. For questions 35 and 36, the diff tool provides partial support. For these questions, in some cases, it might also be possible to apply a code clone detector such as CCFinder [30] to determine a measure of similarity. However, in the cases we observed, the differences were sufficiently large that these tools would be of limited help.

Question 37 (*What is the mapping between these UI types and these model types?*) is an example of a question asked when a programmer develops a (partial) understanding of two related groups of entities and wants to understand the connection between those such as the control-flow between them. Question 40 (*To move this feature into this code what else needs to be moved?*) asks about how a subgraph is connected to the rest of the system. Baniassad and Murphy have developed a technique and a tool, called conceptual module querying, which supports queries about relationships between groups of source code lines, which, in some situations, may help with identifying these connections [2].

Questions 42 (*What will be (or has been) the direct impact of this change?*), 43 (*What will be the total impact of this change?*), and 44 (*Will this completely solve the problem or provide the enhancement?*) are about the impact of (planned) changes to

a system. In situations where an extensive test suite is available, testing allows programmers to determine both if their changes had the desired effect and whether there were any unintended effects [3]. In our studies, comprehensive test suites were not typically available and several of the participants from study two were writing code for an environment that made some types of testing difficult. Various impact analysis techniques and tools such as Fyson and Boldyreff's ripple propagation graph approach [15] and the Chianti test-based tool [47] are intended to help programmers identify the parts of a system impacted by a change. These techniques generate candidates for the programmer to investigate and thus constitute partial support for understanding the impact of changes made.

Questions 38 (*Where should this branch be inserted or how should this case be handled?*), 39 (*Where in the UI should this functionality be added?*), and 41 (*How can we know that this object has been created and initialized correctly?*), as well as several questions discussed previously, such as 26 (*What is the "correct" way to use or access this data structure?*), require understanding a code base at a relatively abstract level. Building this level of understanding based on source code details is difficult and tool support is limited.

Tools or techniques exist that aim to support programmers in recovering a code base's design or architecture through clustering [39], constraint satisfaction [75], or visualization [42]. These techniques aim to support programmers in developing an understanding of the decomposition of a system or its macrostructure and may provide some support for answering questions 26, 38, 39, and 41 to the extent that the answers to these questions can be found along that structure.

Another approach to supporting programmers in abstracting information stems from research around solving the *concept assignment problem*, which is a generalization of the feature location problem discussed above [4]. Tool support for this is limited, though several research tools exist, including DESIRE [4], HB-CAS [17], and PAT [19]. These tools are limited to working with contiguous regions of code and may help with some aspects of understanding code at a higher level. However, many questions in this category (and the previous category) require understanding arbitrary subgraphs, including understanding why things are the way they are and how to use or change things in a way that is consistent with the current code base. Despite some tool support, we believe that developing this level of understanding remains difficult.

6 The Gap

Table 9 summarizes the level of tool support for answering each category of question asked by our participants. We found that questions in the first two categories could all be answered relatively directly using today's tools. The questions in categories three and four required programmers to consider and combine information about multiple points in the source code and were not well supported by tools.

In the studies we conducted, we observed that, when direct tool support was not available for answering a question, programmers resorted to reformulating their initial question as one or more questions that did have tool support available. The process of breaking questions down into ones that were fully (or partially) supported by the available tools was described as *"trying to take my questions and filter those down to something meaningful where I could take a next step"* [N4]. Participant E16 described her (unsuccessful) process of asking a series of questions to answer her higher level question as a path, *"you go down a path to try to find out some information, and it leads to a dead end and you got to start all over again."*

TABLE 9
The Number of Questions with *Full* or *Partial* Support by Category

Question Category	Full	Partial
1. Finding a focus point (5 questions)	3	2
2. Expanding a focus point (15 questions)	12	3
3. Understanding a subgraph (13 questions)	0	13
4. Over groups of subgraphs (11 questions)	0	11
Total	15 (34%)	29 (66%)

Working to answer questions without full tool support resulted in challenges such as dealing with result sets that were noisy when considered in the context of the questions being asked by the participant. In addition to providing information about which questions (or categories of questions) can or cannot be answered directly with today's programming tools, our results also suggest more general limitations with current industry and research tools. Specifically, we believe that programmers need better or more comprehensive support in three related areas: 1) support for more refined or precise questions, 2) support for maintaining context, and 3) support for piecing information together.

6.1 Support for More Refined or Precise Questions

We observed that some questions can be seen as more refined versions of other questions. For example, question 13 (*When during the execution is this method called?*) is more refined than question 12 (*Where is this method called or type referenced?*) and question 33 (*What parts of this data structure are accessed in this code?*) is a more refined version of question 15 (*Where is this variable or data structure being accessed?*). A programmer's questions also often have an explicit or implicit scope. In question 31 (*Which execution path is being taken in this case?*), this is explicit. Question 33 (*What parts of this data structure are accessed in this code?*) asks about changes to a data structure within a certain section of code.

Programmers are often limited in how precise or refined their questions can be. For example, in session 1.11, participant N2 wanted to learn about the properties of an object in a particular iteration of a large loop; getting a handle on this object (in the debugger, for instance) proved difficult. Because tools typically provided only limited support for defining the scope over which to operate, programmers end up asking questions more globally than they intend and, so, the result sets presented will include many irrelevant items. Determining which entities are relevant requires additional exploration.

6.2 Support for Maintaining Context

Most tools are designed to answer a specific kind of question targeting a particular type of artifact and most tools treat questions as if they were asked in isolation. However, in our studies, we have observed that, often, a particular question is part of a larger process involving multiple questions. For example, answering a question may involve gathering information from a code base written in two different languages, each with support from a different set of programming tools. Even when multiple questions are asked using the same tool, the results are presented by that tool in isolation as largely undifferentiated and unconnected lists. Some tools that we have shown to partially help answer higher level questions, such as impact analysis tools, simply produce a list of candidate entities to consider; investigating those can be nontrivial and generally requires using other tools. We believe that there are missed opportunities for tools to make use of the larger context to help programmers more effectively scope their questions and to determine what is relevant to their higher level questions.

6.3 Support for Piecing Information Together

Many tools only support questions involving individual entities and just one type of relationship, though many of the questions asked by our participants go beyond what can be directly asked under these limitations. For example, questions about subgraphs or groups of subgraphs, such as question 30 (*Why isn't control reaching this point in the code?*), require considering multiple entities and relationships. In these situations, as we have discussed, programmers map their questions to multiple tools that they believe will produce information to help answer their question.

At times, we observed that getting accurate answers to a number of questions supported by tools did not necessarily lead to an accurate answer to the original question the participant had in mind. For example, early in session 1.4, one of the participants expressed a desire to *"figure out why one works and one doesn't"* [N4] or, in other words, to compare how the system's handling of one type compared with its handling of a second type. The approach of the pair was to do two series of references searches, one starting from each type under investigation (*"now who calls this method?"* [N6]). Results were compared by toggling between search result sets at each step until the sets first diverged (*"this one only has those two"* [N6]). This point of divergence was taken as an answer to the original higher level question, but, in fact, it was only a partial answer and missed the most important difference. In cases like these, the burden is on the programmer to assemble the information needed to answer their intended question, which can be difficult: *"it gets very hard to think in your head how that works"* [E14] and *"I cannot keep track of all of these similar named things"* [N2]. Tool support is missing for bringing information together, as well as support for building toward an answer.

7 Limitations

The studies that we conducted have allowed us to observe programmers in situations that vary along several dimensions, including the programming tools used, the type of change task, and the level of prior knowledge of the code base. This research approach has allowed us to explore and report on a broad sample of the questions programmers ask, along with behaviors around answering those questions. This focus on breadth also has several limitations.

The use of pairs in our first study likely impacted the change process we observed. We chose this approach to encourage a verbalization of thought processes and to gain insight into the intent of actions performed. An alternative approach to getting similar kinds of information is to have single participants think aloud [67], which was the approach taken in our second study. Although the questions asked in the context of a pair working together may well be different than in the context of an individual working alone, each represents a realistic programming situation.

Our results need to be interpreted relative to the types of tasks used. In the first study, we chose change tasks that could not be completed within the allotted time. We chose complex tasks to stress realism and to stress the investigation of nonlocal unfamiliar code, a common task faced by newcomers to a system and by programmers working on changes that escape the immediate area of the code for which they have responsibility. In the second study, tasks were selected by participants and thus varied significantly. This approach has allowed us to explore a range of realistic tasks. However, not all questions apply to all tasks or to all stages of working on a task and, clearly, our studies do not cover all types of tasks.

In addition to being influenced by the task at hand, the questions asked and the process of answering the questions are influenced by the tools available and by individual differences among the participants themselves. Given a completely different set of tools or participants, our data could be quite different. This fact needs to be considered when interpreting our results or when generalizing them. This limitation is mitigated by three factors. First, our studies cover a range of tools in use today, as well as a range of programmers with different backgrounds and levels of experience. Second, many of the questions we observed programmers asking were independent of the questions that could be answered directly using the tools provided by the environment. Third, we have performed an analysis of tool support for answering questions that covered a wide range of tools.

As mentioned above, the sessions in our two studies varied along several dimensions and we have not analyzed how the questions asked and the answering behavior varied along those dimensions. For example, we have not carefully compared newcomers to a code base with programmers working with code with which they have prior experience nor have we looked for a correlation between the questions asked and the type of tools used. We also have made no effort to rank the questions we observed being asked by some measure of importance and the frequency data we provide in Table 4 is insufficient for

ranking the questions as it does not consider factors such as the time taken to answer a question. Such a ranking would be valuable for prioritizing future research, as well as efforts around building tools to support answering particular questions, but would require a study set up with more controls on the dimensions along which the sessions are allowed to vary.

In both of the studies that we have performed, we observed participants working for only a relatively short period of time (45 minutes for study one and 30 minutes for study two) on tasks that typically required much more time to complete. Similarly, we have not systematically measured how successful our participants were with the task. A follow-up study in which participants were asked to work on a change task to completion would be helpful in at least two ways. First, it would support an analysis of the questions asked at different stages of working on a task. Second, differences in the questions asked and the behavior around answering those questions could be analyzed to determine which approaches tended to be more successful over the course of a task.

Our analysis of the level of tool support for answering questions (presented in Section 5) is limited in two important ways. First, it is possible that we have overlooked a particular research tool applicable to answering one of the questions our participants asked. This limitation means that it is possible that one or more questions have more support than noted in our analysis. Second, as many of the tools that we considered have been evaluated in only limited ways (or not at all in some cases), at times it was difficult to determine the level of support a tool may provide for answering a given question. We believe that these limitations had a small effect on particular details of our analysis and that the overall trends we report hold.

8 Summary

To better understand what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools help in that process, we have collected and analyzed data from two observational studies. Our first study was carried out in a laboratory setting and the second study was carried out in an industrial work setting. Through these studies we have been able to observe a range of programmers working on a range of change tasks using a range of programming tools. Analyzing the data collected from these two studies involved developing generic versions of the questions our participants asked, which slightly abstract from the specifics of a particular situation and code base and analyzing the behavior we observed the participants using to answer those questions. Based on this analysis, we have developed a catalog of 44 types of questions programmers ask and we have categorized those questions into four categories based on the kind and scope of information needed to answer a question.

To understand the degree to which existing tools support answering each type of question, we have analyzed a range of programming tools and techniques for exploring source code information. For each question, our analysis aimed to identify a tool with support for answering that question and to qualitatively evaluate the level of support provided. Based on this analysis, we have identified support that is missing from existing programming tools. Specifically, we found that programmers need better tool support for asking more refined or precise questions, maintaining context, and piecing information together. We also found that tools often target the questions and activities of programmers too narrowly. There is a difference between an environment that provides multiple tools to answer a range of questions and actually supporting a programmer in the process of understanding what they need to know about a software system.

Acknowledgments

The authors are grateful to the participants from their two studies, to Eleanor Wynn for her help with the second study, and to the reviewers for their valuable suggestions. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM, and Intel.

References

[1] B. De Alwis and G.C. Murphy, "Using Visual Momentum to Explain Disorientation in the Eclipse IDE," *Proc. IEEE Symp. Visual Languages and Human Centric Computing,* pp. 51-54, 2006.

[2] E. Baniassad and G. Murphy, "Conceptual Module Querying for Software Engineering," *Proc. Int'l Conf. Software Eng.,* pp. 64-73, 1998.

[3] V.R. Basili and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.,* vol. 13, no. 12, pp. 1278-1296, Dec. 1987.

[4] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster, "Program Understanding and the Concept Assignment Problem," *Comm. ACM,* vol. 37, no. 5, pp. 72-82, 1994.

[5] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l J. Man-Machine Studies,* vol. 18, no. 6, pp. 543-554, 1983.

[6] M.H. Brown, *Algorithm Animation.* MIT Press, 1988.

[7] M.H. Brown, "Zeus: A System for Algorithm Animation and Multi-View Editing," *Proc. IEEE Workshop Visual Languages,* pp. 4-9, 1991.

[8] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," *Proc. 10th Int'l Workshop Program Comprehension,* pp. 241-249, 2000.

[9] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards Understanding Programs through Wear-Based Filtering," *Proc. ACM Symp. Software Visualization,* pp. 183-192, 2005.

[10] T. Eisenbarth and R. Koschke, "Locating Features in Source Code," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 210-224, Mar. 2003.

[11] A. Erdem, W.L. Johnson, and S. Marsella, "Task Oriented Software Understanding," *Proc. Automated Software Eng.,* pp. 230-239, 1998.

[12] K. Erdos and H.M. Sneed, "Partial Comprehension of Complex Programs (Enough to Perform Maintenance)," *Proc. Sixth Int'l Workshop Program Comprehension,* pp. 98-105, 1998.

[13] N.V. Flor and E.L. Hutchins, "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming during Perfective Software Maintenance," *Proc. Fourth Workshop Empirical Studies of Programmers,* pp. 36-64, 1991.

[14] C. Flynt, *Tcl/Tk: A Developer's Guide,* second ed. Morgan Kaufmann, 2003.

[15] M.J. Fyson and C. Boldyreff, "Using Application Understanding to Support Impact Analysis," *J. Software Maintenance Research and Practice,* vol. 10, no. 2, pp. 93-110, Dec. 1998.

[16] B.G. Glaser and A.L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine Publishing, 1967.
[17] N. Gold, "Hypothesis-Based Concept Assignment to Support Software Maintenance," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 545-548, 2001.
[18] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.
[19] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software,* vol. 7, no. 1, pp. 74-81, Jan. 1990.
[20] M. Harman, N. Gold, R. Hierons, and D. Binkley, "Code Extraction Algorithms which Unify Slicing and Concept Assignment," *Proc. IEEE Working Conf. Reverse Eng.,* pp. 11-21, 2002.
[21] A. Hejlsberg, S. Wiltamuth, and P. Golde, *The C# Programming Language,* second ed. Addison Wesley Professional, 2006.
[22] S. Henninger, "Retrieving Software Objects in an Example-Based Programming Environment," *Proc. 14th Int'l ACM SIGIR Conf. Automated Software Eng.,* pp. 408-418, 1991.
[23] J.D. Herbsleb and E. Kuwana, "Preserving Knowledge in Design Projects: What Designers Need to Know," *Proc. Human Factors in Computing Systems,* pp. 7-14, 1993.
[24] R. Holmes and G.C. Murphy, "Using Structural Context to Recommend Source Code Examples," *Proc. Int'l Conf. Software Eng.,* pp. 117-125, 2005.
[25] S. Ibrahim, H.B. Idris, and A. Deraman, "Case Study: Reconnaissance Techniques to Support Feature Location Using Recon2," *Proc. Asia-Pacific Software Eng. Conf.,* pp. 371-378, 2003.
[26] D. Jackson, "Aspect: An Economical Bug Detector," *Proc. Int'l Conf. Software Eng.,* pp. 13-22, 1994.
[27] D. Jackson and E.J. Rollins, "A New Model of Program Dependences for Reverse Engineering," *Proc. SIGSOFT Foundations of Software Eng. Conf.,* pp. 2-10, 1994.
[28] D. Janzen and K. De Volder, "Navigating and Querying Code without Getting Lost," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 178-187, 2003.
[29] W.L. Johnson and A. Erdem, "Interactive Explanation of Software Systems," *Proc. Knowledge-Based Software Eng.,* pp. 155-164, 1995.
[30] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 654-670, July 2002.
[31] A.J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," *Proc. Int'l Conf. Software Eng.,* 2007.
[32] A.J. Ko and B.A. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions about Program Failures," *Proc. Conf. Computer Human Interaction,* pp. 151-158, 2004.
[33] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Trans. Software Eng.,* vol. 32, no. 12, pp. 971-987, Dec. 2006.
[34] J. Koenemann and S.P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," *Proc. SIGCHI Conf. Human Factors in Computer Systems: Reaching through Technology,* pp. 125-130, 1991.
[35] W. Kozaczynski, S. Letovsky, and J. Ning, "A Knowledge-Based Approach to Software System Understanding," *Proc. Knowledge-Based Software Eng. Conf.,* pp. 162-170, 1991.
[36] S. Letovsky, "Cognitive Processes in Program Comprehension," *Proc. Conf. Empirical Studies of Programmers,* pp. 80-98, 1986.
[37] S. Letovsky, "Cognitive Processes in Program Comprehension," *J. Systems and Software,* vol. 7, no. 4, pp. 325-339, Dec. 1987.
[38] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance," *Proc. Conf. Empirical Studies of Programmers,* pp. 80-98, 1986.
[39] S. Mancoridis, B. Mitchell, C. Rorres, Y.-F. Chen, and E. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," *Proc. Int'l Workshop Program Comprehension,* pp. 45-53, 1998.
[40] R. Mays, "Power Programming with the Lemma Code Viewer," technical report, IBM TRP Networking Laboratory, 1996.
[41] N. Miyake, "Constructive Interaction and the Iterative Process of Understanding," *Cognitive Science,* vol. 10, no. 2, pp. 151-177, 1986.
[42] H.A. Muller, M.A. Orgun, S.R. Tilley, and J.S Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification," *Software Maintenance: Research and Practice,* vol. 5, no. 4, pp. 181-204, 1993.
[43] B.A. Myers, R. Chandhok, and A. Sareen, "Automatic Data Visualization for Novice Pascal Programmers," *Proc. IEEE Workshop Visual Languages,* pp. 192-198, 1988.
[44] P.D. O'Brien, D.C. Halbert, and M.F. Kilian, "The Trellis Programming Environment," *Proc. Conf. Object-Oriented Programming, Systems, and Applications,* pp. 91-102, 1987.
[45] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology,* vol. 19, pp. 295-341, 1987.
[46] S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software,* vol. 7, no. 4, pp. 57-66, July 1990.
[47] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," *Proc. Object-Oriented Systems, Languages, Programming, and Applications,* pp. 432-448, 2004.
[48] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Trans. Software Eng.,* vol. 30, no. 12, pp. 889-903, Dec. 2004.
[49] M.P. Robillard and G.C. Murphy, "FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code," *Proc. Int'l Conf. Software Eng.,* pp. 822-823, 2003.
[50] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop, 1980.
[51] B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *Int'l J. Computer and Information Sciences,* vol. 8, no. 3, pp. 219-238, 1979.
[52] J. Sillito, G.C. Murphy, and K. De Volder, "Questions Programmers Ask during Software Evolution Tasks," *Proc. SIGSOFT Foundations of Software Eng. Conf.,* pp. 23-34, 2006.
[53] J. Sillito, K. De Volder, B. Fisher, and G. Murphy, "Managing Software Change Tasks: An Exploratory Study," *Proc. Int'l Symp. Empirical Software Eng.,* 2005.
[54] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," *Proc. IBM Centre for Advanced Studies Conf.,* pp. 209-223, 1997.
[55] V. Sinha, D. Karger, and R. Miller, "Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases," *Proc. Visual Languages and Human-Centric Computing Conf.,* pp. 187-194, 2006.
[56] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.,* vol. 10, no. 5, pp. 595-609, 1984.
[57] R.M. Stallman, "Emacs the Extensible, Customizable Self-Documenting Display Editor," *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation,* pp. 147-156, 1981.
[58] J. Steffen, "Interactive Examination of a C Program with CSCOPE," *Proc. Usenix Winter Conf.,* pp. 170-175, 1985.
[59] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "Jrapture: A Capture/Replay Tool for Observation-Based Testing," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 158-167, 2000.
[60] M.-A.D. Storey, F.D. Fracchia, and H.A. Muller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization," *Proc. Int'l Workshop Program Comprehension,* pp. 17-28, 1997.
[61] M.-A.D. Storey, H.A. Muller, and K. Wong, "Manipulating and Documenting Software Structures," *Software Visualization,* pp. 244-263, 1996.
[62] M.-A.D. Storey, K. Wong, and H.A. Muller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?" *Science of Computer Programming,* vol. 36, nos. 2-3, pp. 183-207, 2000.
[63] A.L. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, 1998.
[64] B. Stroustrup, *The C++ Programming Language,* second ed. Addison Wesley Longman, 1991.
[65] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer,* vol. 14, no. 4, pp. 25-34, Apr. 1981.
[66] D. Tidwell, *XSLT,* first ed. O'Reilly Media, 2001.
[67] M.W. van Someren, Y.F. Barnard, and J.A.C. Sandberg, *The Think Aloud Method; A Practical Guide to Modelling Cognitive Processes.* Academic Press, 1994.
[68] A. von Mayrhauser and A.M. Vans, "From Code Understanding Needs to Reverse Engineering Tool Capabilities," *Proc. Computer-Aided Software Eng.,* pp. 230-239, 1993.

[69] A. Walenstein, "Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework," PhD dissertation, Simon Fraser Univ., 2002.
[70] A. Walenstein, "Theory-Based Cognitive Support Analysis of Software Comprehension Tools," *Proc. Int'l Workshop Program Comprehension,* pp. 75-84, 2002.
[71] M. Weiser, "Program Slicing," *Proc. Int'l Conf. Software Eng.,* pp. 439-449, 1981.
[72] N. Wilde and C. Casey, "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension," *Proc. Working Conf. Reverse Eng.,* pp. 270-276, 1996.
[73] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," *IEEE Software,* vol. 17, no. 4, pp. 19-25, July/Aug. 2000.
[74] D.D. Woods, "Visual Momentum: A Concept to Improve the Cognitive Coupling of Person and Computer," *Int'l J. Man-Machine Studies,* vol. 21, pp. 229-244, 1984.
[75] S.G. Woods, A.E. Quilici, and Q. Yang, *Constraint-Based Design Recovery for Software Engineering: Theory and Experiments.* Springer, 1997.
[76] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 94-106, 2004.

Jonathan Sillito received the BSc and MSc degrees in computing science from the University of Alberta in 1998 and 2000, respectively, and the PhD degree in computer science from the University of British Columbia in 2007. He is currently an assistant professor in the Department of Computer Science at the University of Calgary. His research interests are in the human and social aspects of software engineering. He is a member of the IEEE and the IEEE Computer Society.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. She is currently a professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE and the IEEE Computer Society.

Kris De Volder received the PhD degree from the Vrije Universiteit Brussel, Belgium, in 1998, where his work centered on the use of logic programming to support synthesizing software implemented in Java. He is an assistant professor with the Computer Science Department at the University of British Columbia. He did seminal work on applying logic metaprogramming to aspect-oriented software development. His current research interests are programming-language design and software development tools.