Questions Programmers Ask During Software Evolution Tasks

Jonathan Sillito, Gail C. Murphy and Kris De Volder
Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada
{sillito,murphy,kdvolder}@cs.ubc.ca

ABSTRACT

Though many tools are available to help programmers working on change tasks, and several studies have been conducted to understand how programmers comprehend systems, little is known about the specific kinds of questions programmers ask when evolving a code base. To fill this gap we conducted two qualitative studies of programmers performing change tasks to medium to large sized programs. One study involved newcomers working on assigned change tasks to a medium-sized code base. The other study involved industrial programmers working on their own change tasks on code with which they had experience. The focus of our analysis has been on *what* information a programmer needs to know about a code base while performing a change task and also on *how* they go about discovering that information. Based on this analysis we catalog and categorize 44 different kinds of questions asked by our participants. We also describe important context for how those questions were answered by our participants, including their use of tools.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation

Keywords

Change tasks, development tools, program comprehension, software evolution, empirical study, grounded theory

1. INTRODUCTION

What does a programmer need to know about a code base when performing a change task to a software system? How does a programmer go about finding that information? Many theories have been proposed to describe how programmers may comprehend systems (e.g., [23]) and a range of tools have been built to help programmers with change tasks, such as program databases to answer questions about reference relationships between program elements (e.g., [20]). However, surprisingly little is known about the specific questions asked by programmers as they work on realistic change tasks, and how they use tools to answer those questions.

We are aware of only a small amount of existing work that considers in detail the questions that programmers ask. Erdos and Sneed propose seven kinds of questions programmers must answer while performing a change task based on their personal programming experience [3]. Letovsky presents a taxonomy of questions he observed programmers ask while performing change tasks [11]. Johnson and Erdem studied questions asked to experts on a newsgroup [7].

To provide a more comprehensive and empirically based set of questions that programmers ask during a change task, we undertook two qualitative studies. In each of these studies we observed programmers making source changes to medium (around 20,000 lines of code) to large-sized (over one million lines of code) code bases. The first study was carried out in a laboratory setting with programmers who were new to the code base on which they were asked to work. Some details of this study along with some initial observations were reported previously [15]. This earlier report does not contain an analysis of the particular questions asked by participants, which is a main focus of this paper. The second study was carried out in an industrial work setting with programmers working on code with which they were already familiar. We have used a grounded theory [5, 19] approach to analyze the results from these studies, focusing on the questions that the participants asked as well as their behavior around answering those questions. Our analysis has also considered how they used programming tools in their work on the tasks.

This paper makes two key contributions. The first is an empirically based catalog of the 44 types of questions asked by the participants of the two studies. We have placed these questions into four categories based on the kind of information needed to answer a question: one category groups questions aimed at finding initial focus points, another groups questions that build on initial points, another groups questions that build a model of connected information, and the final category groups questions that integrate across models built from the previous category. To our knowledge this is the most comprehensive such list published to date.

The second contribution stems from observations about the context in which questions were asked and answered. We observed that many of the questions asked were closely related, with some lower-level questions being asked as part of answering higher-level questions. We also observed that the questions a participant asked

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...$5.00.

often mapped imperfectly to questions that could be answered directly using the available tools. Participants, at times, needed to combine result sets or other information from multiple tools to answer their questions. We believe, an understanding of this process of moving from questions to answers provides important context for tool research and design. There is an opportunity to go beyond designing tools to answer a particular kind of question, to designing tools to support programmers more effectively throughout the change process.

We begin with a comparison of our studies to earlier work (Section 2). We then describe our research approach and the setup details for each study (Section 3). Next we describe the questions we observed (Section 4) and issues around answering those questions (Section 5). We then discuss the implications of those results (Section 6), provide a discussion of the limitations of our studies and provide suggestions for follow-up studies (Section 7). We end with a summary (Section 8).

2. RELATED WORK

Various models of program comprehension have been proposed including the top-down model [16], the bottom-up model [13] and the integrated metamodel [22]. These models focus on describing a programmer's mental representation of a program and the cognitive processes and information structures used to form that mental representation. Other work has attempted to use these models or theories to inform tool design. For example, Storey et al. develop a hierarchy of cognitive issues to be considered during the design of a software exploration tool [17].

Our work takes a different approach to influencing the design of tools and we believe our results are complementary to work in the area of program comprehension. 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.

In this section we provide comparisons of our work with previous work considering questions asked by participants (see Section 2.1) studies about programming tool use (see Section 2.2).

2.1 Analysis of Questions

Letovsky presents observations of programmer activities around asking a question, conjecturing an answer and then possibly searching through the code and documentation to verify the answer (i.e., the conjecture) [10, 11]. He reports on five kinds of conjectures: *why* conjectures (questioning the role of a piece of code), *how* conjectures (about the method for accomplishing a goal), *what* conjectures (what is a variable or function), *whether* conjectures (concerned with whether or not a routine serves a given purpose) and *discrepancy* conjectures (questioning perceived discrepancies). The data for Letovsky's taxonomy is from a study of six programmers working on assigned change tasks to a very small program comprising approximately 250 lines of code. In contrast, we aim to develop a more comprehensive list of questions and we aim to do this based on much larger systems, a range of tasks and in the context of the tools available today.

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: (1) where is a particular subroutine/procedure invoked? (2) what are the arguments and results of a given function? (3) how does control-flow reach a particular location? (4) where is a particular variable set, used or queried? (5) where is a particular variable declared? (6) where is a particular data object accessed? and (7) what are the inputs and outputs of a module? [3]. Our work aims to produce a more comprehensive list of questions based on empirical results from a range of participants rather than from our personal experience. Our work also aims to consider higher-level questions than those Erdos and Sneed discuss.

Johnson and Erdem extracted and analyzed questions posted to Usenet newsgroups [7]. They classify these questions 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). By basing this work on newsgroup postings they consider questions asked to experts and they point out that "newsgroup members may have been reluctant to ask questions that should be answerable by examining available code and documentation" [8, page 59]. Our goal has been to identify questions asked during such an examination.

Erdem et al. also analyzed questions from the Usenet study just mentioned along with questions from a survey of the literature (including the work described above) to develop a model of the questions that programmers ask [2]. In their model, a question is represented based on its topic (the referenced entity), question type (verification, identification, procedural, motivation, time or location) and the relation type (what sort of information is being sought). Again, our aim has been to produce a more comprehensive list of questions, including questions at a higher-level than those captured in this work.

Herbsleb and Kuwana report on an empirical study of questions asked by software designers during real design meetings in three organizations [6]. They report on the types of questions asked by their participants as well as how frequently each of those questions were asked. Our approach is similarly empirically based. However we focus on questions asked while performing a change task to a system, rather than questions asked during design meetings for a new system.

2.2 Studies of Tool Use

Storey et al. carried out a user study focused on how program understanding tools enhance or change the way that programmers understand programs [18]. In their study thirty participants used various research tools to solve program understanding tasks on a small system. Based on these results they suggest that tools should support multiple comprehension strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration. In contrast to our work, the Storey et al. study does not attempt to analyze specifically what programmers need to understand.

Three more 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 [14]. Deline et al. report on a formative observational study also focusing on navigation [1]. Our work differs from these in considering more broadly 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 development environment to work on five maintenance tasks on a small program [9]. Their intent was to gather design requirements for a maintenance-oriented development environment. Our studies differ in focusing on more realistic situations 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.

3. STUDY METHOD

This paper reports on a first study carried out in a laboratory setting and a second study carried out in an industrial work setting. Both were observational studies to which we applied qualitative analyses. The participants in the first study (N1...N9) we refer to as *newcomers* as they were working on a code base that was new to them. Participants in the second study (E1...E16) were observed working on code with which they had experience. In both studies the participants were told that the experiment aimed to "learn how programmers manage change tasks." They were *not* told that the questions asked as they performed the change task would be of particular interest. Details of each of these studies are presented in Section 3.1 and 3.2.

To structure our data collection and the analysis of our results, we have used a *grounded theory* approach which has been described as an emergent process intended to support the production of a theory that "fits" or "works" to explain a situation of interest [5, 19]. In this approach, data collection, coding and analysis do not happen strictly sequentially, but are overlapping activities. As data is reviewed and compared, important themes or ideas emerge (called *categories*) that help contribute to an understanding of the situation. As categories emerge, further selective sampling and adaptive coding can be performed to gather more information, often with a focus on exploring variation within those categories. Further analysis aims to organize and understand the relationships between the identified categories, possibly producing higher-level categories in the process.

In our case the primary source of data was audio recordings made during study sessions as well as field notes and some video data. Our analysis was an iterative process of discovering questions in the data and exploring similarities, connections and differences among those questions. In the process we found that many of the questions asked were roughly the same, except for minor situational differences, so we developed generic versions of the questions, which slightly abstract from the specifics of a particular situation and code base. In analyzing those generic questions we 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 the categorization of the questions. We describe these questions and categories in detail in Section 4. This analysis process was similarly applied to the activities we observed around answering questions. In this way we found relationships between some of the questions, including question and sub-question relationships. We also observed important issues for our participants in using the available tools to answer their questions. We describe these issues further in Section 5.

3.1 Study 1

The first study was carried out in a laboratory setting. In this study, pairs of programmers performed change tasks on a moderately sized open source system assigned by the experimenter (the first author of this paper). We choose 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., [4] and [12]).

We carried out a total of twelve sessions. In each session two participants performed an assigned task as a pair working side-by-side at one computer, using the Eclipse Java Development Environment[1] (version 3.0.1) which is a widely used integrated development environment. Following the terminology of Williams et al. [25], 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. The pairings are summarized in Table 1. In the course of this study we observed different pairings working on the same change task, as well as participants working in different pairings.

Table 1: Session number (SN), driver and observer for each session of the first study.

SN	Driver	Obs.	SN	Driver	Obs.
1.1	N7	N3	1.7	N3	N1
1.2	N4	N1	1.8	N7	N5
1.3	N4	N7	1.9	N8	N6
1.4	N6	N4	1.10	N8	N2
1.5	N5	N3	1.11	N9	N2
1.6	N5	N2	1.12	N9	N6

All nine participants 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; all except N4 and N9 had two or more years. All participants in this study were initially newcomers to the code base used.

In each session, the programming pair was given forty-five minutes to work on a change task. The tasks assigned to participants 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 comprises roughly 60KLOC. The tasks were complex, completed tasks chosen from ArgoUML's issue-tracking system (issues 484, 1021, 1622 and 2718). We did not expect that the participants would be able to complete the tasks in the time allotted, but we believed that they would be able to make significant progress. Participants were asked to accomplish as much as possible on the assigned task, but not to be concerned if they could not complete the task. Participants were stopped after the forty-five minutes elapsed regardless of how much progress had been made. No effort was made to quantify how much of the task had been completed. In four of the sessions (sessions 1.4, 1.7, 1.11 and 1.12) participants were asked to work on the same task that they had worked on in a previous session, allowing us to gather data about later stages of work on the task.

During each session an audio recording was made of discussion between the pair of participants, a video of the screen was captured, and a log was made automatically of the events in Eclipse related to navigation and selection. At the end of the session the experimenter, who was present during each session, 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. An audio recording of each interview was made.

[1] `http://www.eclipse.org`, last verified July 2006

[2] `http://argouml.tigris.org`, last verified July 2006

Table 2: Session number (SN), participant(s) (Part.) and the primary programming languages and tools for each session of the second study.

SN	Part.	Primary languages/tools
2.1	E1	C++, TCL/Emacs, DDD, Virtual desktops
2.2	E2	C++, TCL/Emacs (split window)
2.3	E3	C#, XSLT/Visual Studio, BizTalk Orchestration
2.4	E4	C#/Visual Studio, BizTalk Orchestration
2.5	E5	C#, ASP/Visual Studio
2.6	E6,E7	C#, ASP/Visual Studio, NetMeeting
2.7	E8	Java/Netbeans
2.8	E9	SQL, MDX/Enterprise Manager, Query Analyzer, etc
2.9	E10	C++, Batch/Notepad, Visual Studio
2.10	E11	C/Proprietary loading and debugging tools (embedded)
2.11	E12	C, C++/Visual Studio (two instances)
2.12	E13	HTML/UltraEdit, Proprietary Document Manager
2.13	E14	C/Emacs (split window)
2.14	E15	XML, Java/VIM (two instances)
2.15	E16	C/VI (two instances), GDB

3.2 Study 2

We carried out a second study with 16 programmers in an industrial setting. In this study we studied individual programmers working alone (rather than in pairs) because that was their normal work situation. The one exception was participants E6 and E7 who were accustomed to working together and who participated in the same session, as they normally would work. Each of these sessions (numbered 2.1 to 2.15) is summarized in Table 2. A brief description of each of these tools is provided in the appendix.

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 during the sessions the participants used the tools they would normally use. For example participant E1 worked on a C++ and TCL code base using tools such as Emacs and DDD, while E3 worked on a C# and XSLT code base using Visual Studio.

We asked each participant to select the task on which they worked to ensure that the tasks were realistic and because we were interested in observing programmers working on a range of change tasks. The participants were asked, in advance of each session, to select a task that would be "involved, not a simple local fix". Beyond that no guidance was given. In each session the participant was asked to describe their task selection and then to spend about thirty minutes working on that task. They were asked to think-aloud while working on the task [21]. After thirty minutes the experimenter (same as for the first study) interviewed the participant about the session. The interviews were informal and focused on the challenges faced and their use of tools. An audio recording was made and field notes were taken during each session, including during the interview portions of the sessions.

4. RESULTS

Our analysis has focused on the questions our participants asked about the source code on which we observed them working. We report on 44 kinds of questions we observed our participants asking. These questions are generalized versions of the specific questions asked by our participants. For example, N4 asked the question *"How does [MAssociation] relate to [FigAssociation]?"* which is reported below as *How are these types or objects related?* (see question 22). As participants were not always explicit about their questions, in some cases we inferred an implicit question based on their actions and comments.

We organize the discussion of these questions around four categories: questions about finding initial focus points, questions about building on such points, questions aimed at understanding a subgraph, and questions over multiple subgraphs. Considering a code base as a graph of entities (methods and fields, for example) and relationships between those (references and calls, for example), to answer any given question requires considering some subgraph of the system. The properties of that subgraph are the basis for our categorization as illustrated in Figure 1. Questions in the first category are about discovering an initial entity in the graph. Questions in the second category are about a given entity and other entities directly related to it. Questions in the third category are about understanding a number of entities and relationships together. Questions in the final category are over such connected groups; how they relate to each other or to the rest of the system, for example.

Although other categorizations of the questions are possible, we have selected this categorization for three reasons. First, the categories show the types and scope of information needed to answer the given questions. Second, they capture some intuitive sense of the various levels of questions asked. Finally, the categories make clear various kinds of relationships between questions, as well as certain challenges in answering questions as described in Section 5.

The order in which we present the categories is not representative of how the questions were necessarily asked. In fact we observed that participants often jumped around between various activities or explorations, at times leaving questions only partially answered, 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)).

4.1 Finding Initial Focus Points

One category of questions asked by our participants, the newcomers from the first study in particular, focused on finding initial 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). 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 starting point. These are perhaps similar to what Wilde and Casey call "places to start looking" [24].

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). Similarly 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). The following is a summary of the types of questions asked in this category. Each question is followed by a list of the sessions in which we observed it being asked.

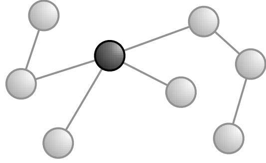
1. Which type represents this domain concept or this UI element or action? (1.1 1.2 1.3 1.5 1.6 1.7 1.8)

2. Where in the code is the text in this error message or UI element? (1.5 1.9)

Finding initial focus points

5 kinds of questions.

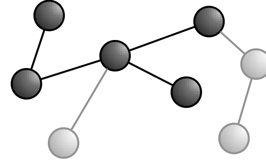
For example: Which type represents this domain concept?



Building on those points

15 kinds of questions.

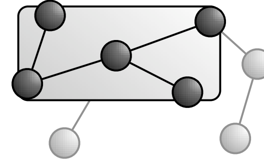
For example: Which types is this type a part of?



Understanding a subgraph

13 kinds of questions.

For example: What is the behavior these types provide together?



Questions over groups of subgraphs

11 kinds of questions.

For example: What is the mapping between these UI types and these model types?

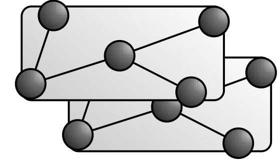


Figure 1: An overview of the four categories of questions asked by our participants. Each is illustrated by a diagram depicting source code entities along with connections between those entities.

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)
4. Is there a precedent or exemplar for this? (1.1 1.10 1.12 2.6 2.14 2.15)
5. Is there an entity named something like this in that unit (for example in a project, package or class)? (1.1 1.2 1.4 1.5 1.6 1.10)

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. For example, E16 used grep from the command line to find an exemplar for what she needed to do. On the other hand, 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 (for example, Eclipse's Package Explorer, which provides a tree view of the packages and classes in a system) was used. At times the number of search results or candidates 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 running the application in debug mode 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 Building on Those Points

A second category of questions were about building from 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 [i.e., 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.

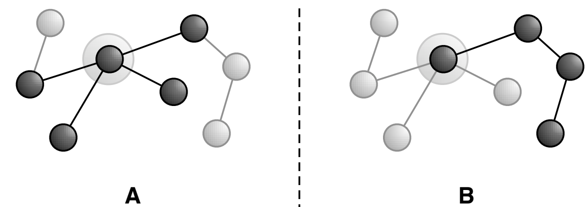


Figure 2: A depiction of two observed patterns of questions: (A) multiple questions about the same entity, and (B) a series of questions where each subsequent question is about a newly discovered entity.

Sometimes we observed a series of questions about the same entity, forming a star pattern as depicted in part A of Figure 2 (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 part B of Figure 2.

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). The following summarizes these kinds of questions along with a list of the sessions during which each question was asked.

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? (1.5 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); and *"Who is using the factory?"* (N4).

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)

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 1.5 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 may be about a call to a particular method. The distinction is that with 12 the participant is asking for all callers, while with 13 the participant is asking about a particular caller, such as might be discovered by looking at the call stack in a debugger.

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 arguments (*"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 2.11 2.13 2.15)

18. What are the arguments to this function? (1.2 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 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 were 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, 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)

24. What in this structure distinguishes these cases? (1.2 1.12 2.8)

25. What is the behavior these types provide together and how is it distributed over the types? (1.1 1.2 1.3 1.4 1.6 1.11 1.12 2.11)

26. What is the "correct" way to use or access this data structure? (1.8 2.15)

27. How does this data structure look at runtime? (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. For example: *"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 isn't control reaching this point in the code? (1.4 1.9 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)

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)

We observed that to answer these questions, participants often revisited entities, repeating questions such as those described in Sections 4.1 and 4.2; *"I forgot what we figured out from that"* (N3); *"I want to have another look at this guy"* (N8). Part of the issue is that for a participant to see or discover relevant entities and relationships individually was not always sufficient to mentally build an answer to the questions in this category; *"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? Why is the namespace null? We added that"* (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?*) presented above is about understanding a particular flow through a number of methods, while question 34 presented in this section 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). 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); *"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; such as question 37, which 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? (1.2 1.3 1.4)
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 1.10 1.11 2.6 2.11 2.14 2.15)
37. What is the mapping between these UI types and these model types? (1.1 1.2 1.5)

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 frame work, 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; for example: *"There's a lot of the interactions between the different modules that aren't exactly understood"* (E10). 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)
40. To move this feature into this code what else needs to be moved? (2.7 2.13)
41. How can we know 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 1.8 1.10 1.11 1.12 2.1 2.7 2.12 2.15)
43. What will be the total impact of this change? (1.7 2.1 2.3 2.4 2.5 2.9 2.11)
44. Will this completely solve the problem or provide the enhancement? (1.1 1.9 1.11 2.2 2.14)

5. QUESTIONS IN CONTEXT

To this point we have described specific kinds of questions asked by our participants organized around the four categories summarized in Figure 1. We have also described some behavior around answering specific questions. This section aims to put these questions and activities into context by describing aspects of the observed process of asking and answering questions. This process is summarized in Figure 3. Specifically, we found three important considerations. First, the interaction between a participant's questions, with some questions being asked as part of answering other questions (see Section 5.1). Second, the interaction between the questions our participants asked and those that the available tools could answer (see Section 5.2). Finally, taking the results produced by the tools and using those to answer a participant's intended question (see Section 5.3). We end this discussion with a more detailed anecdote from the second study that provides an overview of this process and many of the issues raised (see Section 5.4).

5.1 Questions and Supporting Questions

We observed that many of the questions asked in our studies were closely related. At times an answer to a higher-level question was pursued by asking a number of other, lower-level questions. In fact many of the lower-level questions (especially those from the second category) were clearly in support of higher-level questions. For example in session 1.4, the participants asked the question *"What classes are relevant?"* (N6). This question was not directly supported by any of the tools available; instead the participants asked several other questions around finding candidate classes and building on those results. As candidate classes were found, the participants set break points and ran the application to answer the question *"is this the thing"* (N6). This process was repeated until several relevant entities had been discovered. During the interview following session 1.4 participant N6 described the process as exploration to come up with a hypothesis (i.e., a candidate class) and then checking that hypothesis.

On the other hand, the process we observed was not always obviously driven by a higher-level question. At times the lower-level questions were asked first. For example in session 1.2 the participants began by asking various questions such as *"Do you see something that seems like a 'name'?"* (N1) and *"Does [this type] have a guard?"* (N1). Answering these questions gave them various pieces of information about several relevant types and relationships. Only later did they ask questions such as *"I am kind of curious how this class integrates with this whole hierarchy"* (N1). While attempting to answer this question, one participant expressed *"I am getting lost in the details"* (N1), where the details were lower-level questions and the answers to those. Though in such cases the lower-level questions came first, they can still be

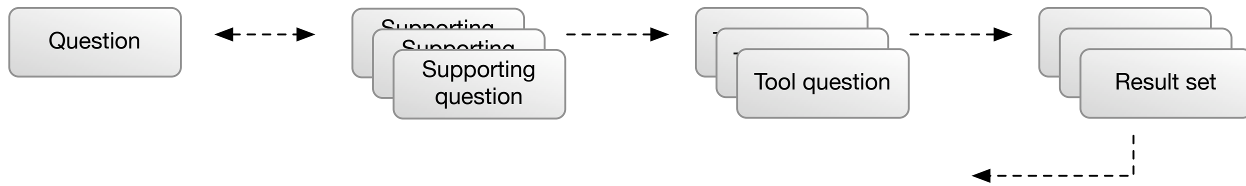


Figure 3: An illustration of the process of using the tools available to answer questions about a software system. This figure depicts the relationships between questions, supporting questions, tool questions and results from tools.

regarded as being in support of the higher-level questions. Answering these first questions produces information that both motivates and helps answer the later questions.

5.2 Questions and Tools

In various ways participants used the tools available to them to answer their questions. This mapping between questions and tools was more successful in some situations than others. At times the tools used answered a question somewhat different than the one our participants were trying to answer. Wanting to know *"Which classes have MEvents as fields?"* (N3), or as the other participant expressed it *"So we want to see what kinds of things have MEvents?"* (N5), the participants in session 1.5 performed a references search in Eclipse. The search returned 102 results (obviously including more kinds of references than they had in mind), of which they looked at only a few, before abandoning that line of inquiry.

Due to a mismatch between the participant's intentions and the questions supported by tools, the same approach could prove more or less effective in various situations. For example, in session 1.2 the participants wanted to find a *"representation of this transition class"* (N1). Using the Open Type tool in Eclipse they asked the question *which types have 'transition' in the name?*, which worked well. However, a similar approach failed for the participants in session 1.3 who looked for types with 'association' in the name, because the list was large and provided no way for the participants to differentiate the results (i.e., no information in the result list indicated what was relevant).

At other times the participants appeared to choose a tool or an approach that was not optimal. For example, in session 1.12 the participants spent several minutes reading code trying to figure out the flow of data in a pair of methods and in particular the value of one of the variables (*"So [variable 1] is [variable 2] once you've parsed out the guard and the actions?"* (N6)). This was time consuming and by the end of their exploration they were left with incorrect information. If they had used Eclipse's debugger (which was available to them) they could have very quickly and accurately answered the question.

There were also times when there seemed to be no tool that could provide much direct assistance. For example E2's task was to do a merge between two versions of a pair of files (a C++ source file and header file). *"The changes were overlapped enough that diff is completely confused as to what parts should merge [...] It is showing all these differences that are just in the wrong parts of the code"* (E2). The result was that he used a very manual approach (using a split Emacs window) to answer his questions about what areas to merge. This involved *"jumping between the source and the header trying to compare what was moved out and hoping that the compiler will catch the syntax errors because of code that was inserted improperly"* (E2).

5.3 From Results to Answers

The lack of a direct mapping between the questions our participants pursued and the questions supported by tools (the tool questions might be *"too general"* (N7), for example) had a number of consequences. One was that the results produced by tools were often noisy when considered in the context of the questions being asked by the participant. This is true even for tools that provide no false positives relative to the questions that they are designed to answer.

We observed that, at times, getting accurate answers to a number of questions that could be posed to the tools did not necessarily lead to an accurate answer to the question the participant had in mind. For example, early in the 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 participants' approach was to do two series of references searches, one starting from each of the types under investigation (*"Now who calls this method?"* (N6)). Results were compared by toggling between search result sets at each step until they first diverged (*"This one only has those two"* (N6)). This point of divergence was taken as an answer to their higher-level question, but in fact it was only a partial answer and missed the most important difference.

In some situations piecing together the results from a number of queries was problematic: *"I can't keep track of all of these similar named things"* (N2); *"we were retracing steps we had done before and [weren't] aware of it"* (N1). And naturally there where times when the participants wanted some kind of overview, rather than narrow sets of search results: *"I think I would need some kind of overview document that says... this is the architecture of how the thing works and the main classes involved"* (N9); *"I think we really have to get a wider view"* (N3).

5.4 A More Involved Anecdote

We end this section with one more anecdote from the second study. It provides an overview of the process participant E16 carried out to answer her questions, and demonstrates that at times multiple supporting questions must be pursued and that multiple tools are used to answer those questions. It also demonstrates that working in this way, while producing a significant volume of re-

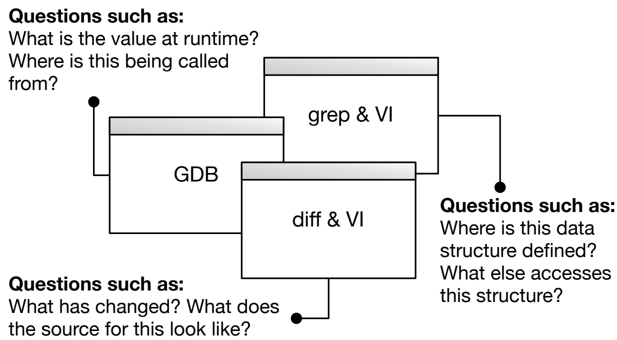


Figure 4: Participant E16's arrangement of windows and associated tools, along with the types of questions asked in each window.

sults, does not necessarily result in an answer to the original question posed. In these cases, the participant must often 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.

Central to the task E16 worked on was a connected group of complicated data structures (*"kinda complicated, there are a lot of enums and variants, if it is that variant then these fields apply and if it is that variant than those fields apply"*). She spent a significant portion of the session trying to determine how to get a certain value from an instance passed to a particular routine (see questions 26 and 27 described in Section 4.3). To determine this she asked many lower-level questions around callers and especially the parts of data structures. Activities around asking and answering these questions were interleaved, with the goal of building an answer to the higher-level question. She used several different tools (GDB, diff, grep and VI). Each of these tools were in separate windows, arranged as in Figure 4. She explained her arrangement in this way: *"I got 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. I try to always arrange them in the same way so that I can remember where I am at as I windows switch."*

She described her process 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."* She had mapped her question down in a certain way, pursued a number of lower-level questions, but by the end of the session she had failed to answer her question: *"So now I have to think, what is another way I can figure out what the variable is from the field"*.

6. IMPLICATIONS OF OUR RESULTS

While further research is needed to work through the full implications of these results, in this section we cover a few initial observations connecting our results to tool design. We observed participants struggle to refine their questions and to then map those questions to tools. One reason for this is that most tools support only relatively low-level questions. For example, tools generally only support questions about one entity and type of connection; while some of the questions articulated by our participants were at the level of groups of entities and connections of different types. Also, generally tools are designed to answer a particular kind of questions and so multiple tools were often used to answer a participant's higher-level question. None of the tools used by our participants helped with this refinement or helped maintain the context of the higher-level question being asked.

At times, the mapping between a participant's questions and the questions supported by the selected tool were imperfect. This often caused the result sets from tools to include many items irrelevant to the original questions (e.g., false positives relative to the information the participant was seeking). Determining relevance required additional exploration on the part of the participant. This imperfect mapping between questions and tools together with the use of multiple tools required participants to mentally piece together pieces of information from multiple (often noisy) result sets. Results were presented by tools in isolation as largely undifferentiated and unconnected lists with no support for *building* towards an answer to a participant's question.

These results provide important context for tool research and development. We believe that tool design has often targeted the questions and activities of programmers too narrowly, which has resulted in tools that answer very narrow questions. Further, there is a difference between an environment providing multiple tools that answer a range of questions and actually supporting a programmer in the process of understanding what they need to know about a software system. Our results point to the need to move tools closer to programmers' questions and the process of answering those questions.

7. DISCUSSION

The studies that we have performed 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 which we discuss in Section 7.1.

In presenting our results we have provided some limited frequency data describing the frequency of question types across sessions. This data is summarized in Figure 5 which shows the distribution of observed question occurrences across the two studies by category. An occurrence consists of one or more questions from a category being asked in a session. This data illustrates that questions in the first three categories occurred more frequently during the first study than the second, while for the fourth category the breakdown between studies was more even. One possible reason for the differences is that participants in the fist study were newcomers to the code they were working on while participants in the second study were working with code they had experience with, though other factors may have also contributed. In Section 7.2 we discuss ways to build on our work by addressing open research questions such as what factors precisely contribute to these differences in question frequency.

7.1 Limitations of Studies

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 [21], 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, both represent realistic programming situations.

Our results need to be interpreted relative to the types of tasks used. In the first study we choose change tasks that could not be completed within the allotted time. We chose complex tasks to stress realism and to stress the investigation of non-local unfamil-

Finding initial focus points

Study 1 (75%) Study 2 (25%)

Building on those points

Study 1 (67%) Study 2 (33%)

Understanding a subgraph

Study 1 (67%) Study 2 (33%)

Questions over groups of subgraphs

Study 1 (48%) Study 2 (52%)

Figure 5: The distribution of observed question occurrences across the two studies by category. An occurrence consists of one or more questions from a category being asked in a session. For example, 25% of the sessions that featured a question from the first category were from study two.

iar 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 so 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 those 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 needs to be considered when interpreting our results or when generalizing them. This is mitigated by the fact that our studies cover a range of tools in use today as well as a range of programmers with different backgrounds and levels of experience. Also, 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. This suggests that some of our results will likely generalize to other tool sets.

7.2 Future Studies

The two studies we have performed and analyzed have provided a wealth of information around programmers asking and answering questions. At the same time it has left several related and important research questions unanswered. Here we discuss several of these open questions and suggest how future studies could build on our work.

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. Similarly we have not looked for a correlation between the questions asked and the type of tools used. Various follow-up studies to support these comparisons are possible. Studies which vary one of the possible factors (the task, the tool set or the experience level of the participants, for example) and fixed the others could support such comparisons.

We have made no effort to rank the questions we observed being asked by some measure of importance. Such a ranking would be valuable for prioritizing future research as well as efforts around building tools to support answering particular questions. Although the occurrence data that we have provided could provide a basis for such a ranking, we believe that this would be insufficient. Instead, a study that collected and analyzed data around how much time was spent on particular questions as well as how successful programmers were at answering their various questions should provide a more effective ranking of questions.

In both of the studies that we have performed, we observed participants working for only a relatively short period of time (forty-five minutes for study one and thirty minutes for study two), on tasks that required much more time typically to complete. 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.

8. SUMMARY

From our observational studies of the information programmers need to understand during a change task, we have identified 44 different kinds of questions being asked. We found that these questions could be categorized into four categories based on the characteristics of the source code graph capturing the information needed for answering a given question: those aimed at finding initial focus points, those aimed at building on such points, those aimed at understanding a subgraph, and those over such subgraphs. To our knowledge this is the most comprehensive catalog of questions published to date.

In addition to describing specific questions asked by our participants, we have also described some of the behavior we observed around answering those questions, including their use of tools. Our analysis exposed several important contextual issues around the relationships between questions, mapping questions onto programming tools, and using the results from those tools to answer the intended question or questions. We believe that these results provide important insights into the difference between providing a tool to answer a particular question and truly providing the support that programmers need to complete a task.

Acknowledgments

The authors are grateful to the participants from our two studies, to Eleanor Wynn for her valuable help with the second study, and to Brian de Alwis for his comments on an early draft of the paper. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM and Intel.

9. REFERENCES

[1] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of ACM 2005 Symposium on Software Visualization*, pages 183–192, 2005.

[2] A. Erdem, W. L. Johnson, and S. Marsella. Task oriented software understanding. In *Proceedings of Automated Software Engineering*, pages 230–239, 1998.

[3] K. Erdos and H. M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *Proceedings of 6th International Workshop on Program Comprehension*, pages 98–105, 1998.

[4] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *Proceedings of the Empirical Studies of Programmers: Fourth Workshop*, pages 36–64, 1991.

[5] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing, 1967.

[6] J. Herbsleb and E. Kuwana. Preserving knowledge in design projects: What designers need to know. In *Proceedings of Human Factors in Computing Systems (CHI)*, pages 7–14, 1993.

[7] W. L. Johnson and A. Erdem. Interactive explanation of software systems. In *Proceedings of Knowledge-Based Software Engineering (KBSE)*, pages 155–164, 1995.

[8] W. L. Johnson and A. Erdem. Interactive explanation of software systems. *Proceedings of Automated Software Engineering*, 4(1):53–75, 1997.

[9] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 126–135, 2005.

[10] S. Letovsky. Cognitive processes in program comprehension. In *Proceedings of Conference on Empirical Studies of Programmers*, pages 80–98, 1986.

[11] S. Letovsky. Cognitive processes in program comprehension. *The Journal of Systems and Software*, 7(4):325–339, 1987.

[12] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10:151–177, 1986.

[13] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[14] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[15] J. Sillito, K. D. Volder, B. Fisher, and G. C. Murphy. Managing software change tasks: An exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering*, 2005.

[16] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE10(5):595–609, 1984.

[17] M.-A. Storey, F. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems, special issue on Program Comprehension*, 44(3):171–185, 1999.

[18] M.-A. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[19] A. L. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for developing Grounded Theory*. Sage Publications, 1998.

[20] W. Teitelman and L. Masinter. The interlisp programming environment. *IEEE Computer*, 14:25–34, 1981.

[21] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg. *The Think Aloud Method; A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

[22] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of CASE'93*, pages 230–239, 1993.

[23] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[24] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of WCRE*, pages 270–276, 1996.

[25] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, 17(4):19–25, 2000.

APPENDIX

Tool Descriptions

We have observed programmers using a wide range of programming tools. For reference, we briefly describe these tools.

- Eclipse Java Development Environment: Integrated Java development environment.
- Visual Studio: Integrated development environment with support for a range of programming languages including C/C++, C#, and J#.
- Netbeans: Integrated Java development environment providing basic static analysis and debugging support.
- GNU Project Debugger (GDB): A command-line debugger for languages such as C and C++.
- Data Display Debugger (DDD): A graphical user interface for command-line debuggers (such as GDB).
- NetMeeting: A multipoint video conferencing application. Supports sharing desktops.
- Virtual Desktops: Software allowing users to organize application windows into multiple contexts.
- BizTalk Orchestration: Visual programming tool for describing business processes.
- Enterprise Manager: Administrative tool for Microsoft SQL Server.
- Query Analyzer: Tool for authoring and analyzing queries to execute against Microsoft SQL Server.
- Emacs, VI, VIM and UltraEdit: Basic source code editors.