Asking and Answering Questions During a Programming Change Task

by

Jonathan Sillito

B.Sc., The University of Alberta, 1998
M.Sc., The University of Alberta, 2000

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

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

December 2006

Abstract

Despite significant existing empirical work, little is known about the specific kinds of questions programmers ask when evolving a code base. Understanding precisely what information a programmer needs about the code base as they work is key to determining how to better support the activity of programming. The goal of this research is to provide an *empirical foundation for tool design based on an exploration of what programmers need to understand about a code base and of how they use tools to discover that information.* To this end, we undertook 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 to 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 a systematic 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 four key contributions: (1) a catalog of 44 types of questions programmers ask, (2) a categorization of those questions into four categories based on the kind and scope of information needed to answer a question, (3) a description of important context for the process of answering questions, and (4) a description of support that is missing from current programming tools.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I am grateful to the participants from my two studies, to Eleanor Wynn for her valuable help with the second study, and to Gail Murphy for assistance above and beyond the call of a committee member. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM and Intel.

Dedication

To Christina Jolayne for her patient and enthusiastic support through far too many years of school.

Chapter 1

Introduction

To better support the activity of programming there has been substantial research on tools to make programming more effective (e.g., [6, 23, 40, 62]). Despite this research and despite the many commercial and research programming tools developed, programming remains a difficult activity. We believe that tools have not helped as much as they might because much of the work on tools is based on intuition or personal experience rather than empirical research results. In fact software engineering research papers in general seem at times to proceed based on various relatively unsubstantiated assumptions about the support that programmers need from their tools or about the challenges with which programmers need assistance. For instance, here are several examples of assumptions made without empirical backing: "being forced to follow multiple different relationships... results in developers losing their context" [87]; "one of the problems most frequently faced in dealing with legacy software is the location of the code for a specific feature" [108]; and "in design recovery we need to be able to generate a range of views mined from both structural and behavioral information about the code" [75].

Our intention here is not to claim that these statements are necessarily incorrect, nor is it to claim that the tools developed are without value, we simply wish to illustrate that much of this work is not guided by empirical data. We also believe that a more solid, empirical foundation is possible and that such a foundation would be valuable for tool builders. For example, we believe empirical results can produce additional insights beyond intuition and allow researchers to address the right problems and to do so in a manner more likely to be useful to practitioners.

A large body of significant empirical work about programming activities does exist. Some of this work has focused on developing models of program comprehension, which are descriptions of the process a programmer uses to build an understanding of a software system (e.g., [83, 53]). One of the goals of work on program comprehension has been to inform the design of programming tools (e.g., [102, 92]). There have also been several studies about how programmers perform change tasks including how programmers use tools in that context (e.g., [18, 86]).

Despite this work many open questions remain. For example, 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? Although this is of central importance to the activity of programming and the design of programming tools, 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. For example, we are aware of only a small amount of existing work that considers in detail the questions that programmers ask. While studying programmers performing change tasks, Letovsky observed a recurring behavior of programmers asking questions and conjecturing answers and he detailed some of those questions [57]. Erdos and Sneed proposed seven kinds of questions programmers must answer while performing a change task [17]. This list was based on their personal programming experience and includes questions such as *where is a particular subroutine/procedure invoked?* Johnson and Erdem studied questions asked to experts on a newsgroup [47]. Similarly, little is known about how programmers answer their questions and the role of tools in that process.

Our aim is to build on this body of existing work with the goal of providing an *empirical foundation for tool design based on an exploration of what programmers need to understand and of how they use tools to discover that information.* We focus, in particular, on what programmers need to understand about a code base while performing a nontrivial change task to a software system. To this end, 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. Based on a systematic analysis of the data from these user studies, this dissertation makes four key contributions: (1) a catalog of 44 types of questions programmers ask, (2) a categorization of those questions into four categories based on the kind and scope of information needed to answer a question, (3) an analysis of the process of answering questions which exposed important context for the questions, and (4) an analysis of existing tool support for answering questions including a discussion of the support that is currently missing from tools. These contributions provide a foundation on which to build tools that more effectively support programmers in the process of performing a change task, and in particular gaining an understanding of a code base.

In the remainder of this chapter we present an overview of our studies and our research approach (Section 1.1), describe our four key contributions (Section 1.2) and outline the contents of this dissertation (Section 1.3).

1.1 Overview of Our Data Collection and Analysis

We collected data from two studies: our first study was carried out in a laboratory setting [85] and the second study was carried out in an industrial work setting [84]. Both were observational studies to which we applied qualitative analysis. 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. All nine participants in the first study were computer science graduate students with varying amounts of previous development experience, including experience with the Java programming language. In this first study, pairs of programmers performed change tasks on a moderately sized open-source system assigned by the experimenter. 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. This study involved twelve sessions (1.1...1.12) with each participant participating in two or three sessions with different pairings in each session.

The second study entailed 15 sessions (2.1...2.15) carried out with 16 programmers (E1...E16) in an industrial setting. With one exception, in this study we studied individual programmers working alone (rather than in pairs) because that was their normal work situation. 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. 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. They were asked to think-aloud while working on the task [101]. 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. Varying the study situation along these dimensions has been deliberate as we believe that a broad look at the process of asking and answering questions was needed.

To structure our data collection and the analysis of our data, we have 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 [25, 96]. Grounded theory analysis revolves around various coding procedures which aim to identify, develop and relate the concepts. In this approach, data collection, coding and analysis do not happen strictly sequentially, but are overlapping activities. As data is reviewed and compared (a process referred to as "constant comparison"), important themes or ideas emerge (i.e., *categories*) that help contribute to an understanding of the situation. As categories emerge, further selective sampling 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. 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.

During the sessions of both studies, including the interview portions of those sessions, the experimenter (the author of this dissertation) made field notes. These field notes were later supplemented by transcripts of much of the audio data. Summaries rather than full transcripts were made of several portions of our audio data (in particular from the second study) that did not relate to our analytic interests. Our coding work proceeded from these transcripts, though also with occasional support of the audio data. Three kinds of coding were used in our analysis: open coding which involves identifying concepts in the data (as opposed to assigning predefined categories), selective coding which focuses on further developing identified categories, and theoretical coding which focuses on exploring the relationships between categories. Coding was primarily performed by the author of this dissertation, however members of the supervisory committee (Gail Murphy, Kris De Volder and Brian Fisher) supervised this work.

Coding of our empirical data proceeded in four major phases, each of which involved a relatively detailed review of nearly all data collected to the point in time at which the coding was performed. The first phase involved open coding of only the data from the first study, and was conducted both during and after that study. The results of this coding are summarized in Section 3.2. The goal of this phase was not to achieve saturation, but simply to help provide a general understanding of how our participants managed their work and help identify central issues for further analysis. Based on this first phase of coding we decided to focus our further analysis on our participants' questions and their activities around those questions.

In the second phase, which began before the second study commenced and continued during and after that study, we used selective coding to identify the questions asked by our participants. This phase also involved "constant comparison" of 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 very slightly abstract from the specifics of a particular situation and code base. For example, we observed specific comments from session 2.15 including *"I am just looking at the data structures here trying to figure out how to get to the part I need to get to"* [E16] and *"to fix it I have to somehow get the variable from the field to pass into this function"* [E16]. Abstracted versions of the questions this participant was asking are *What data can we access from this object?*

and *How can data be passed to (or accessed at) this point in the code?* (see questions 16 and 28 discussed in Chapter 5).

The questions identified are low-inference representations of our data which have (along with the transcripts) formed a basis for theoretical coding which focused on further comparisons of the questions asked. In the process 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 a categorization of these questions into four categories discussed further in Section 5.1. During this third phase we also selectively coded for activities around answering the identified questions both at the particular question level and at the category level. A final phase of analysis involved a review of both the audio data and the transcripts to insure the consistency of our coding. This also gave us confidence that no significant questions or other insights relative to our analytic interests were missed, and that the analysis could stop.

Beyond the analysis of our empirical data we have also considered the level of tool support for answering questions provided by a wide range of tools, including relevant tools discussed in the research literature. This investigation drew on our findings around the questions asked and the behavior we observed around answering those questions. The goal here was to explore the level of support today's tools provide for answering the questions we have identified and to increase the generalizability of our results.

1.2 Overview of Contributions

Based on our systematic analysis of the data collected from the two studies, this research makes four key contributions that contribute to an understanding of what programmers need to know about a code base and the role tools play in discovering that information. The work described in this thesis is covered in two publications [85] and [84].

1. An empirically based catalog of the 44 types of questions asked by the participants of the two studies. These are slight abstractions over the many specific questions and situations we have observed. These abstractions have allowed us to compare and

contrast the questions asked as well as to gather some simple frequency data for those questions. Example questions include: *Which types is this type a part of?* which was asked in two different sessions, *What in this structure distinguishes these cases?* which was asked in three different sessions, and *Where should this branch be inserted or how should this case be handled?* which was asked in seven different sessions. Although not a complete list of questions, it illustrates the sheer variety of questions asked and provides a basis for an analysis of tool support for answering questions. To our knowledge this is the most comprehensive such list published to date.

2. A categorization of the 44 types of 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 over one or more models built from the previous category. 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 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 observed challenges around answering those questions.

3. An analysis of the observed process of asking and answering questions, which provides valuable context for the questions we report. We have shown that many of the questions asked were closely related. For example, some lower-level questions were asked as part of answering higher-level questions. We have shown that the questions a participant asked often mapped imperfectly to questions that could be answered directly using the available tools. For example, at times the questions programmers pose using current tools are more general than their intended questions. We have also shown that programmers, at times, needed to mentally combine result sets or other information from multiple tools to answer their questions. These results contribute

to our understanding both about how programmers answer their questions and the challenges they face in doing so.

4. An analysis of the support existing tools (both industry tools and research tools) provide for answering each kind of question. We consider which questions are well supported and which are less well supported. We also generalize this information to demonstrate a gap between the support tools provide and that which programmers need. In particular we show that programmers need improved support for asking higher-level questions and more precise questions, support for maintaining context and putting information together, and support for abstracting information and working with subgraphs of source code entities and relationships. We hope these results will provide motivation and a foundation for the design of future programming tools.

Our analysis has allowed us to consider our data at different levels, each producing different insights. We believe that these insights will be of value to both research and tool design. To show this, this dissertation also includes a discussion of how tool research and design can benefit from our work (i.e., an exploration of the implications of these results).

1.3 Overview of Dissertation Contents

Chapter 2 (Related Work)

Chapter 2 compares our research to previous work, including work in the area of program comprehension (including, work that has proposed cognition models, efforts to use those models to inform tool design and work that analyzes questions asked by programmers) and empirical studies of how programmers manage change tasks.

Chapters 3 and 4 (User Studies)

Chapters 3 and 4 give setup details (tasks, participants, tools used, etc.) for the two studies we performed. The laboratory study is described in Chapter 3. The industry study is described in Chapter 4. These chapters also discuss some initial observations from each study. Our initial observations from the first study are organized around eight key observations and five challenges. Our initial observations from the second study focus on aspects of the situation that differ from that of the first study.

Chapter 5 (Questions in Context)

The results of our analysis of the data collected from our two studies is presented in Chapter 5. Broadly, this presentation is in two parts. The first is in Section 5.1 and presents the 44 types of questions along with our categorization of these questions into four top-level categories. The second is in Section 5.2 and presents context around answering those questions.

Chapter 6 (Analysis of Tool Support)

Further analysis of these questions, including a discussion of how well existing industry and research tools support a programmer in answering these questions, is presented in Chapter 6. Support for each question is rated as *full* (i.e., well supported), *partial* (i.e., some support but not complete), and *minimal* (i.e., little or not tool support). Chapter 6 also contains an overall discussion of the type of tool support that is currently lacking.

Chapter 7 and 8 (Discussion and Summary)

Chapter 7 discusses possible implications for our results including making specific suggestions for programming tool design. Chapter 7 also contains a discussion of the limits of our results and a discussion of possible future studies that could be used to build on our work. We conclude this dissertation with a summary in Chapter 8.

Chapter 2

Related Work

In this chapter, we discuss a range of work related to our research. For each of these we describe the similarities and differences with our work. We cover work in the area of program comprehension including cognitive models, efforts to use those theories to inform tool design, and studies around the analysis of programmers' questions (see Section 2.1). We also cover previous empirical studies that have looked at how programmers use tools and generally how they carry out change tasks and other programming activities (see Section 2.2). This coverage includes a discussion of studies that use similar research methods. However a discussion of particular research tools to support various programming activities is not provided here. Instead, a detailed discussion of this work is provided in Chapter 6, where we analyze the support a wide range of research and industry tools provide for answering the questions programmers ask.

2.1 Program Comprehension

Program comprehension or software understanding is a broad field encompassing a large body of work. Some of this work has focused on proposing cognitive models. We discuss this work in Section 2.1.1. Other work has attempted to use these models or theories to inform tool design. This work is discussed in Section 2.1.2. A third category of work in this area that is most closely related to our work, has focused on analyzing the questions programmers ask. This work is discussed in Section 2.1.3.

2.1.1 Cognitive Models

A cognitive model describes the cognitive processes and information structures used by programmers to form a *mental model*, which is a programmer's mental representation of the program being maintained. Many cognitive models have been proposed, and there are both similarities and differences between these models [103]. For example all models rely on the programmer's own knowledge, the source code and available documentation [103]. Disparities between various comprehension models can be explained in terms of differences in experimental factors (programmer characteristics, program characteristics and task characteristics) that influence the comprehension process [91].

In this section we discuss several key cognitive models organized around three categories: top-down program comprehension, bottom-up program comprehension, and models combining multiple strategies.

Theories by Brooks [8, 9], Koenemann and Robertson [53], and Soloway and Ehrlich [88] are all based on a *top-down* approach. 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. This process starts with a hypothesis about the top-level goal of the program which is then refined by forming sub-hypotheses. This process then continues recursively (in a depth-first manner) until the hypotheses can be verified. Brooks claims that verification depends on beacons in the code, which are features that give clues that a particular structure or operation is present. Again, experimental support exists for each of these models, however the generalizability of the results from the supporting experiments may be limited.

According to the *bottom-up* theory of program comprehension, 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 [83]. This process is sometimes referred to as *chunking* and is driven by the limitations in short term memory [61]. Two theories that propose a bottom-up approach are Pennington's model [71] and Shneiderman and Mayer's cognitive framework [82]. Pennington claims that programmers use a bottom-up approach to form two kinds of models. The first is the program model, which is a model of the control-flow of the program. The second is the situation model, which is a model of the data-flow and functions of the program. Shneiderman and Mayer's framework considers syntactic knowledge and semantic knowledge separately, with semantic knowledge being built in a bottom-up fashion. Each of these proposed models are supported by empirical evidence, although the evidence is based on experiments involving small programs.

Littman *et al.* noted that programmers use either a *systematic* strategy or an *as-needed strategy* [58]. Their experiments found that programmers need to get both static knowledge (structural information) and causal knowledge (information about runtime interactions between components) and that using an as-needed approach made the second type of information difficult to gain accurately. Letovsky's *knowledge-based* understanding model proposes that programmers work "opportunistically", using both bottom-up and top-down strategies [56, 54]. A core component of this model is a knowledge base that encompasses a programmer's expertise and background knowledge. Soloway *et al.* also propose a model in which programmers use a number of different strategies in the process of trying to understand a program including: inquiry episodes (read, question, conjecture and search cycle), systematic strategies (tracing the flow of the program) and as-needed strategies (studying particularly relevant portions of a program) [89]. Finally, von Mayrhauser and Vans propose a model that they call the *integrated metamodel*. This model combines four components. Three of these describe the comprehension process: the top-down model (i.e., a domain model), the program model (a control-flow abstraction) and the situation model (a data-flow and functional abstraction). The fourth component of the integrated metamodel is a supporting knowledge base as in Letovsky's model.

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 often abstracted away by theories of comprehension. To do this we have thoroughly analyzed the information that programmers need to discover. We have also analyzed programmers' behavior around discovering that information along with the role that tools play in the answering process.

2.1.2 Informing Tool Design

One of the goals of work in the area of program comprehension has been to inform the design of programming tools (or better documentation methods [89]) to support a programmer with various program understanding processes. As a primary goal of our research has also been to empirically support the design of programming tools, in this section we discuss several efforts in this direction.

Von Mayrhauser and Vans' approach to this is based on their proposed integrated metamodel [102]. Specifically they have used their model (and the supporting empirical studies) to produce a list of *tasks* and *subtasks* programmers need to perform as part of understanding a system. For each of these they have suggested associated *information needs* along with *tool capabilities* to support those information needs. For example, a programmer building a program model (a task) will need to investigate and revisit code segments (a subtask). To do this he or she will need access to previously browsed locations (an information need) which could be supported by a tool that provides a history of browsed locations (a tool capability).

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 [92]. Their framework is based on a wide range of program comprehension work and is inspired by a similar framework from the domain of hypermedia tools [99]. Elements in this hierarchy represent possible design goals (for example, *enhance top-down comprehension*) with the leaves capturing more concrete design goals (or features) a tool should have given those higher-level design goals (for example, *support goal-directed hypothesis-driven comprehension* and *provide an adequate overview of the system architecture at various levels of abstraction*). Storey *et al.* suggest that this hierarchy, along with an iterative design, implement and evaluate approach to tool building can lead to tools that effectively support programmers in exploring code [93].

Work by Walenstein represents a different approach to bridging the gap between cognitive models and tool design [105, 104]. His goal is to provide a more solid theoretical grounding for the design of programming tools based on program comprehension theories as well as other cognitive theories. To this end, Walenstein discusses a *theory of cognitive support.* While a program comprehension theory describes the process of building a mental model, a 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 that support might be based on 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 begin with models of cognition or other cognitive theories, we begin with observations about how programmers manage a change task and develop 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.

2.1.3 Analysis of Questions

Possibly the research that is closest to our work is previous research into the questions programmers ask or the information they need to perform their work. Like our work most of this research (the work by Erdos and Sneed being the one exception) is based on the analysis of qualitative data collected from studies of programmers performing change tasks.

Letovsky presents observations of programmer activities which he calls *inquiries* [56, 57]. These may involve a programmer asking a question, conjecturing an answer and then possibly searching through the code and documentation to verify the answer (i.e., the conjecture). Letovsky believes there are five kinds of conjectures (and therefore five kinds of associated questions). The first three kinds or categories of conjectures are: *why* conjectures (questioning the role of a piece of code), *how* conjectures (about the method for accomplishing a goal) and *what* conjectures (what is a variable or function). The last two categories interact with the first three. They are *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 (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 realistic 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? [17]. Note that these are seven specific questions, rather than a number of categories as presented by Letovsky. Our work aims to produce a more comprehensive list of questions, but based on empirical results from a range of participants. Our work also aims to consider higher-level questions than those discussed by Erdos and Sneed.

Johnson and Erdem extracted and analyzed questions posted to Usenet newsgroups [47]. 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). By basing this work on newsgroup postings they were looking at 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" [48, page 59]. Our goal has been to identify questions asked during such an examination.

Building on this work, Erdem *et al.* also analyzed questions from the Usenet study just mentioned and questions from a survey of the literature (including the work described above) to develop a model of the questions that programmers ask [16]. In their model, a question is represented based on its topic (the referenced entity), the question type (one of: verification, identification, procedural, motivation, time or location) and the relation type (what sort of information is being asked for). 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 have empirically studied questions asked by software designers during real design meetings in three organizations [38]. They determined the types of questions asked as well as how frequently they were asked. Based on a separate study they also present questions asked by programmers concerning software requirements [55]. 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 or during requirements gathering.

2.2 Empirical Studies of Change Tasks

The situation of programmers performing change tasks has been studied from a number of perspectives. Many of these have, at least partially, 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 [95]. In their study thirty participants used various research tools to solve program understanding tasks on a small system. Based on these 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.

Murphy *et al.* report on observations around how programmers are using the Eclipse Java Development Environment [67]. These observations are based on usage data from 41 programmers collected by an Eclipse plugin that monitors a wide range of user actions. Observations include the percentage of programmers using the various tools (the Package Explorer, the Console and the Search Results View were used by the highest percentage), the commands used by the most programmers (Delete, Save and Paste were among those used by the most programmers), and which refactoring commands were used by the most programmers (Rename being the most popular). These results should provide important information for tool builders on how their tools are being used. We used a similarly instrumented version of Eclipse in our first study, which has allowed us to collect some of this same type of data. However we have collected data over a much shorter duration which limits the conclusions we can draw. Instead we have focused our analysis efforts on the qualitative data we have collected.

More similar to our study are efforts that qualitatively examine the work practices of programmers. For example, Flor *et al.* used distributed cognition to study a single pair of programmers performing a straightforward change task [18]. We extend their methods to 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 [86]. 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 [77]. Deline *et al.* report on a formative observational study also focusing on navigation [15]. Our study 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 [51]. 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 software developers experience disorientation when using the Eclipse Java integrated development environment [1]. They analyzed their data using the theory of visual momentum [111], 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.

2.3 Summary

Our work is related to work in the area of program comprehension. Much of this research has focused on proposing and validating cognitive models, that is models of how programmers understand programs. Other work has attempted to use those models to inform the design of tools. Most closely related to our work in this area are previous efforts around analyzing the questions programmers ask as the perform a change task. Our work aims to build on this previous work by providing a much more comprehensive list of questions, by analyzing programmer behavior around answering those questions and also by analyzing tool support for answering questions. Our work is also related to previous work that empirically studies how programmers perform change tasks or how programming tools are used. We differ from much of this work by studying more realistic situations and by focusing more specifically on the process of asking and answering questions.

Chapter 3

Study 1: Laboratory-based Investigation

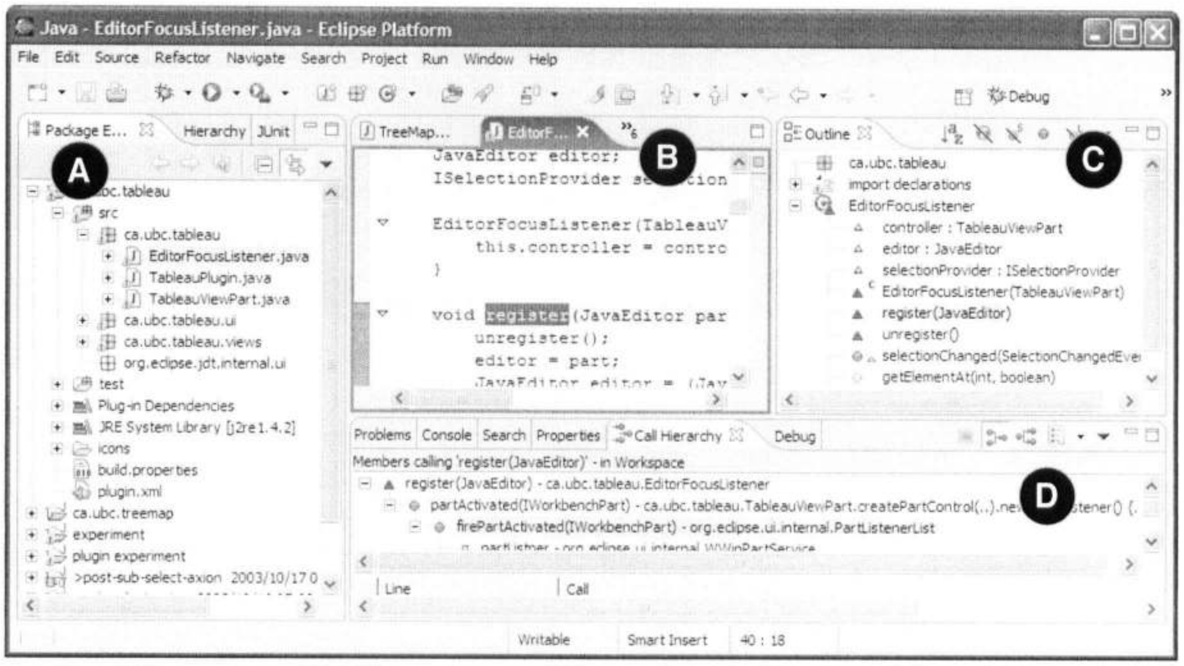
The first study we carried out was conducted in a laboratory setting. The goal of this study was to observe programmers performing significant change tasks using state-of-the-practice development tools. Several study design choices were made to make this as realistic as possible. Specifically we used: real change tasks, experienced programmers and a non-trivial code base (about 60KLOC). The study involved nine participants (all of whom were computer science graduate students) and a total of twelve sessions. 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., [18] and [63]). 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 (the author of this dissertation), who was present during each session, briefly interviewed the participants about their experience.

In this chapter we present study details for this first study, including: setup (Section 3.1), tasks (Section 3.1.1) and participants (Section 3.1.2). We also cover some initial observations that resulted from our analysis of the data from this study only. These initial observations capture the basic activities we observed organized around eight key observations which are summarized in Table 3.3. Examples of these key observations include *Goals were initially* *narrowly focused, but became more broad in scope as programmers struggled to understand the system sufficiently to perform the task.* (see Observation 2) and *Revisiting entities and relationships was common, but not always straightforward.* (see Observation 8). As described earlier (see Section 1.1) we have used a grounded theory approach in analyzing our data. The observations presented here are based on the categories we developed as part of trying to understand the situation under study and before focusing our analysis more closely on the question and answering process. In addition to being interesting in their own right, presenting these initial observations provides insights into the way our theoretical understanding of this situation has developed as we applied grounded theory analysis.

We selected complex, non-local changes for our participants to work on. The reason for this choice was to make the situation realistic and to gather data about a challenging situation, one that would likely benefit from tool support. As expected, the participants in this study found the assigned tasks challenging and many participants expressed a feeling of having made little progress during a session or specific parts of a session. In this chapter we present our initial impressions of the major challenges that the participants faced in completing the tasks: *gaining a sufficiently broad understanding* and *ineffective use of tools*, for example. Results from a more extensive analysis of the data from both studies, including further discussion about these challenges, is presented in Chapter 5.

3.1 Study Setup

Participants in this study used the Java programming language [28] and the Eclipse Java development environment (version 3.0.1), a widely used IDE that we consider representative of the state-of-the-practice [43]. A screenshot of Eclipse is shown in Figure 3.1 showing several commonly-used views or tools: the package explorer (showing the package, file and class structure), the tabbed source code editor, the content outline view (showing the structure of the currently open file) and the call hierarchy browser. Other commonly-used views not shown in the screenshot include a type hierarchy view, a search results view (showing results from both lexical and static analysis based searches), a breakpoint view, a variable view


Java - EditorFocusListener.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Package E... Hierarchy JUnit
A
ca.ubc.tableau
src
ca.ubc.tableau
EditorFocusListener.java
TableauPlugin.java
TableauViewPart.java
ca.ubc.tableau.ui
ca.ubc.tableau.views
org.eclipse.jdt.internal.ui
test
Plug-in Dependencies
JRE System Library [j2re1.4.2]
icons
build.properties
plugin.xml
ca.ubc.treemap
experiment
plugin experiment
>post-sub-select-axion 2003/10/17 0
TreeMap... EditorF...
B
Outline
C
ca.ubc.tableau
import declarations
EditorFocusListener
controller : TableauViewPart
editor : JavaEditor
selectionProvider : ISelectionProvider
EditorFocusListener(TableauViewPart)
register(JavaEditor)
unregister()
selectionChanged(SelectionChangedEve
getElementAt(int, boolean)
Problems Console Search Properties Call Hierarchy Debug
D
Line Call
Writable Smart Insert 40 : 18

```
    JavaEditor editor;
    ISelectionProvider se    cion

    EditorFocusListener(TableauV
        this.controller = contro
    }

    void register(JavaEditor par
        unregister();
        editor = part;
        JavaEditor editor = (JavV
```

Members calling 'register(JavaEditor)' - in Workspace
register(JavaEditor) - ca.ubc.tableau.EditorFocusListener
partActivated(IWorkbenchPart) - ca.ubc.tableau.TableauViewPart.createPartControl(..).ne   stener() {.
firePartActivated(IWorkbenchPart) - org.eclipse.ui.internal.PartListenerList
partListner - org.eclipse.ui.internal.WWinPartService

Figure 3.1: A screenshot of the Eclipse Development Environment, with several major tools labeled: (A) package explorer, (B) source code editor with tabs, (C) content outline view, and (D) call hierarchy browser.

(showing runtime values of variables in scope while executing an application in debug mode), and a launch view (which shows the execution stack, also while in debug mode).

The study involved twelve sessions (1.1...1.12). In each session two participants performed an assigned task as a pair working side-by-side at one computer. Following the terminology of Williams *et al.* [110], 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 3.1.

In each session, the programming pair was given forty-five minutes to work on a change 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

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 3.1: Session number, driver, observer and assigned task for each session.

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. 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. The experimenter, 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. An audio recording of the interview was made.

3.1.1 Change Tasks

Table 3.2 describes the change tasks assigned to participants. The tasks were all enhancements or bug fixes to the ArgoUML[1] 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. Table 3.2 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 at times multiple smaller changes are committed at the same time. However, these estimates illustrate that these tasks were based on complex, non-local changes, which we selected to make the situation realistic and to gather data about a challenging situation, one that would likely benefit from tool support.

Because of the choice of tasks, we did not expect that the participants would be able to complete the tasks 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 one task, but not to be concerned if they could not complete the task. In four of the sessions, 1.4, 1.7, 1.11 and 1.12, participants were asked to work on the same task they had worked on in a previous session, allowing us to gather data about later stages of work on the task. Table 3.1 shows which tasks were assigned for each session.

3.1.2 Participants

Nine programmers (N1...N9) participated in our first study. 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

[1]`http://argouml.tigris.org`, last verified August 2006

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.
2718	13	Fix a model saving error that occurs after a use case with extends relationships is deleted from the model.

Table 3.2: Study tasks along with an estimate on the number of files needed to be changed to perform the task. Numbers refer to IDs in the ArgoUML issue tracking system. Note that task 1622a is not an ID from the tracking system, as the task is simply a variation on task 1622.

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. Each participant participated in two or three sessions (see Table 3.1). All participants in this study were initially newcomers to the code base used. The participants worked on the same code base for each session and may have gained some familiarity with the code base over the course of those sessions.

3.2 Initial Observations

We begin with a brief summary of the actions we observed participants perform and the tools they used. Following this summary we present some initial results from our analysis of the data collected in this study. These results are organized around eight observations (summarized in Table 3.3). These observations are based on the categories and dimensions we identified as being important to the situation under study. The method by which we developed these observations is described in more detail in Section 1.1. The description of each observation includes supporting data, often in the form of quotes from the recorded dialog, as well as comments on related research where possible. In Section 3.2.1 we present several observations around our first impressions on the challenges faced by the participants in this study. Note that the results presented in this chapter have been published previously [85].

The actions we observed may be summarized into five categories: (1) static exploration of the source code using a number of different tools, including viewing the source code in the editor; (2) setting break points and running the application in debug mode, which allows stepping through the execution and inspecting the execution stack and object values; (3) making paper notes; (4) making changes to the source code; (5) and reading online API documentation. Figure 3.2 shows the specific tools that were used during static exploration and debugging activities. The percentages in the figure represent the proportion of logged events (ignoring selections in the editor) over all sessions using the given tool. Various navigation actions, such as navigate to declaration, account for an additional 11.7% of the

1	Goals were often decomposed into sub-goals that could be investigated directly, but the sub-goals were not always easy to form.
2	Goals were initially narrowly focused, but became more broad in scope as programmers struggled to understand the system sufficiently to perform the task.
3	Programmers wrote code early in the change process.
4	Programmers minimized the amount of code that was investigated in detail.
5	Exploration activities were of two distinct types: (1) those aimed at finding initial focus points and (2) those aimed at building from such points.
6	Building a complete understanding of the relevant code was difficult.
7	Programmers' false assumptions about the system were at times left unchecked and made progress on the task difficult.
8	Revisiting entities and relationships was common, but not always straightforward.

Table 3.3: Summary of key observations.

logged events and are not shown in the figure. This data indicates that the programmers did make use of many of the facilities of the development environment.

Observation 1: **Goals were often decomposed into sub-goals that could be investigated directly, but the sub-goals were not always easy to form.**

In all of the sessions, participants decomposed goals (or questions) into sub-goals that could be more directly supported by the tools in the development environment. N4 described this process as *"trying to take [my] questions and filter those down to something meaningful where I could take a next step"* [1.3]. As an example of this process, early in 1.10 the programmers determined that they needed to understand where in the code call events were being parsed. To accomplish this goal, four sub-goals or sub-questions were identified whose resolution were relatively directly supported by the development environment.

1. *"Find out where [`MCallEvent`] gets created"* [N8]. The action for this sub-goal was a reference search on the specified class; this query returned two results which the

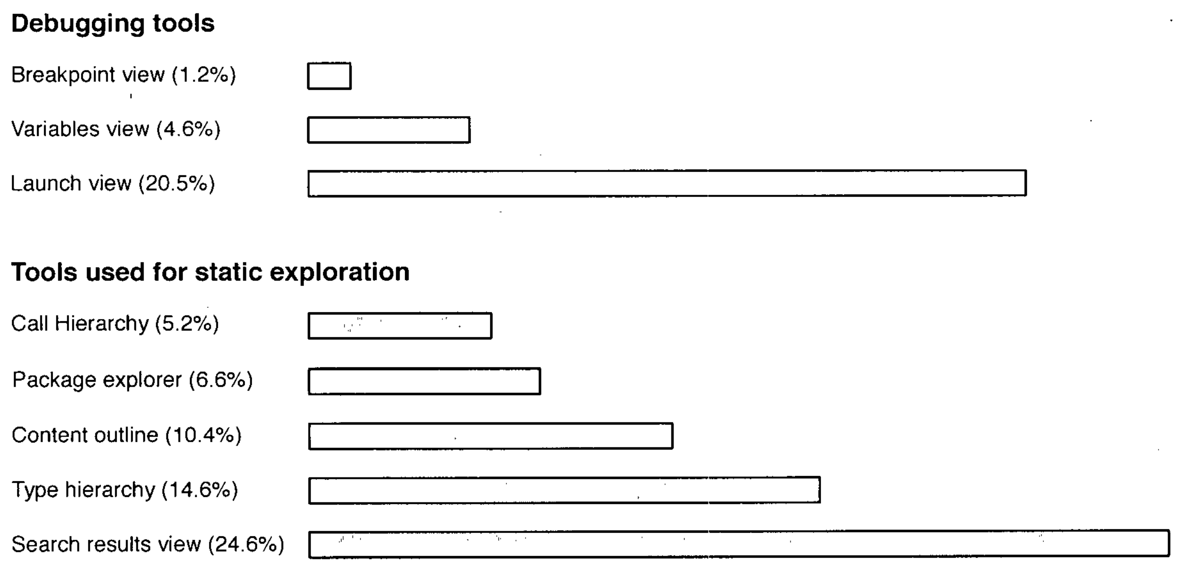


Figure 3.2: A summary of tool usage over all sessions. The percentages represent the proportion of recorded events using the given tool. Only those tools that accounted for at least 1% of the events are shown. Navigational events accounted for an additional 11.7% of the events.

participants inspected in the search results view. They decided that *"[MFactoryImpl] is a good place to start"* [N8].

2. Find out where the event object creation (`MFactoryImpl`) is initiated during the parsing of text input. From the search view, the programmers opened the `createCallEvent` method of the `MFactoryImpl` class and then searched for references to the method. They then followed a chain of three more searches for references directly in the search view, eventually resulting in them opening the `parseEvent` method from the `ParserDisplay` class in the editor.

3. Verify that the method found was in fact where the parsing takes place: *"do you think [parseEvent] is it?"* [N8]. To accomplish this sub-goal the participants set a break point and ran the application in debug mode. They were able to confirm their hypothesis when the breakpoint was hit.

4. Finally they wanted to find an appropriate point on the execution path to begin making changes to the code: *"so we have to search backwards from here"* [N8]. They used the variables view (in Eclipse's debug perspective) to determine various stages in the parsing.

Translating goals in this way was not always straightforward. For example, we observed that the question *"how does [MAssociation] relate to [FigAssociation]?"* [1.3/N4] can not be answered directly in the development environment, but rather requires a significant amount of exploration to identify and integrate the various relationships involved. While this behavior bears some resemblance to the top-down model of program comprehension described in Section 2.1.1, we see this as driven by the level at which the available tools operate.

Observation 2: **Goals were initially narrowly focused, but became more broad in scope as programmers struggled to understand the system sufficiently to perform the task.**

At the start of a change task, participants in this study typically attempted to learn as little as possible about the system, focusing on very specific parts of the system, rather than learning about broader issues, such as the package structure or architecture of the system. For instance, the pairs in sessions involving task 1622 (1.2, 1.4, 1.5 and 1.7) all focused on understanding how control reached the `PropPanelCallEvent` class to learn some key behavior assumed important for the task, rather than working to understand overall how the panel GUI worked.

For several sessions (most notably 1.6, 1.7, 1.9 and 1.12), the participants felt by the end of the session that to succeed with the task they needed to consider the system more broadly: *"I would definitely need a big picture, we are sort of in this small little bit of code [...] we need to back up further and see what else was out there"* [1.12/N9]. This comment seems to have been caused by a realization that the solution for the task is more complex than first suspected. As another example, at the end of session 1.7, the driver felt that *"focusing on how to solve the task is too premature, because we're never going to figure it out if we are too narrow, I think we really have to get a wider view"* [N3].

One approach suggested by our participants as a means of gaining a broader view was to essentially continue the same exploration approach but getting gradually more broad (*"we're kind of circling in a spiral [...] we got to get a bit further out"* [N1]). This suggestion is consistent with the bottom-up approach described in Section 2.1.1, though in the sessions we observed, participants aimed to develop an understanding of specific task relevant concerns in the source code, rather than an entire program. Three other approaches were suggested by our participants: looking at architectural documentation, exploring the package structure, and using a *"brute force"* [1.6/N2] approach, which involved stepping through the running application using the debugger, looking at much more of the system than had been considered up to that point.

Observation 3: **Programmers wrote code early in the change process.**

In half of the sessions (1.1, 1.5, 1.7, 1.8, 1.10 and 1.12) programmers felt that a relatively narrow and incomplete understanding of the relevant code was sufficient to begin making changes to the source code. In these sessions, programmers began writing code as soon as they knew enough to begin. The pair in session 1.5 referred to their writing of code as *"mucking around with things"* [N3]. In contrast, in the other six sessions, the programmers set out to gain an understanding of all (or most) of the relevant code before beginning to make any changes. One programmer expressed discomfort coding too soon: *"I think I have just seen too many cases where [programmers] never come back and it is just really hard to maintain it afterwards"* [1.8/N7].

When coding occurred it appeared to be part of an exploratory process that served several purposes that appear to have not yet been discussed in the literature. First, it minimized the amount of information that participants needed to remember as many pieces of information that were learned could be captured explicitly in the code. Second, it served as a way to check assumptions, especially when combined with the use of Eclipse's debugger. For example, while writing code the programmers in session 1.5 kept the target application running in debug mode and continually switched between coding and inspecting the system in the debugger to ensure that the code they had written was executing when and how they expected it would. Finally, writing code helped support a narrow investigation of the system (Observation 2) as only those parts of the system needed to write the code had to be understood.

Observation 4: **Programmers minimized the amount of code that was investigated in detail.**

In addition to being narrow in their investigation of the code (Observation 2), the participants tended to avoid looking at source code in detail. For example, in 1.2 the observer said to the driver *"if I were you I would click on the interfaces, because the classes which implement it will have a lot of detail that is not so important"* [N1]. Similarly, the participants in 1.4 when considering the `MEvent` hierarchy (Figure 3.3) initially ignored the implementations of the classes and the interfaces and simply looked at the relationships involved, beginning with the type hierarchy.

In general, the participants appeared reluctant to look closely at the source code for an entity—a class or method—until after they had developed an initial understanding of the entity using tools that provided an abstract view of it, and until the participants felt that it was sufficiently important to their task. The participants in 1.3 were the most obvious exception to this rule as they spent a relatively large amount of time reading source code, which appeared to be detrimental as little progress was made on the task. The more common and successful behavior we observed is consistent with the observation by Robertson *et al.* that programmers do not read source code line by line [76].

Also, programmers tended not to systematically explore more than about three search results. In some cases, rather than explore a large number of results the programmers would attempt to refine their query, although producing sufficiently precise queries was often difficult. In other cases the results were disregarded entirely.

Observation 5: **Exploration activities were of two distinct types: (1) those aimed at finding initial focus points and (2) those aimed at building from such points.**

Activities that focused on identifying relevant information can be divided into two sets: (1) those aimed at finding initial focus points (or "places to start looking" [109]) and (2) those aimed at building from such points. N2 said that much of his effort in 1.10 involved finding initial focus points: *"that was the main point, finding that spot where you can focus in on"* [N2].

As an example of an effective approach to finding initial starting points, the programmers in sessions 1.5 searched for possibly relevant entities and set break points at some of these places. The application was then run in debug mode to see which of the identified points were in fact along the relevant control path. This approach gave the pair immediate feedback on their hypotheses, and confidence in what they had found. In session 1.9 the programmers began looking for such a point in the code by performing text searches based on an error message.

Given a relevant point (or points) to focus on, programmers often changed their approach and began building from that point, using several different means: references searches (*"let's see if [targetChanged] gets called"* [1.6/N5]), opening the entity in the type hierarchy (*"what does [NavPerspective] inherit from?"* [1.1/N3]), stepping through the method in the debugger, and reading the source code. Similarly, Ko *et al.* report that once relevant code had been identified, programmers explored the code's dependencies [51].

Several times, programmers had difficulty identifying the information that they needed to perform the task. Sometimes the information could not be found because the tools were not helpful; other times the programmers did not make effective use of the tools. Tools

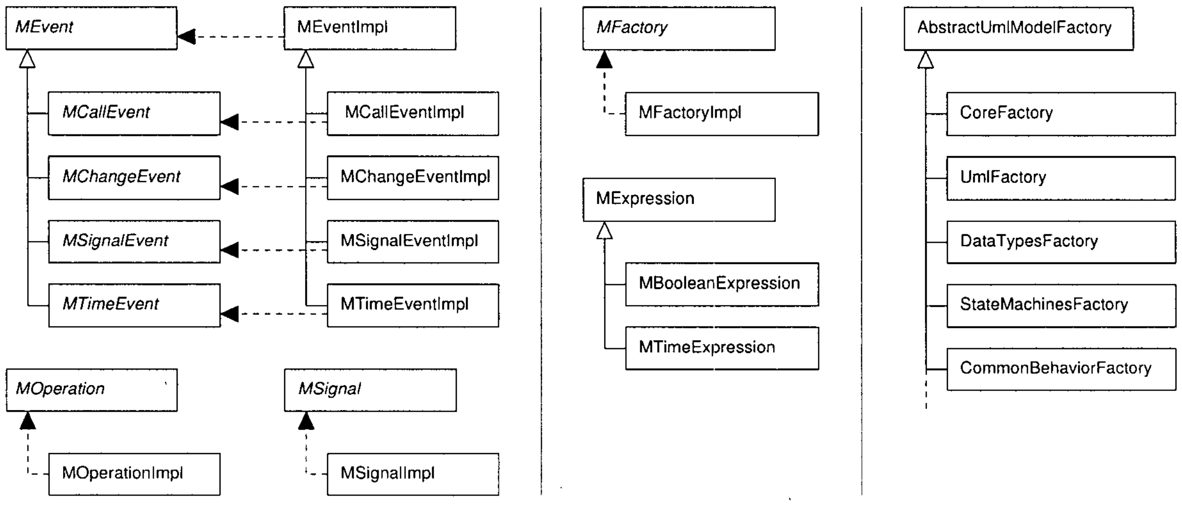


Figure 3.3: The model classes and interfaces (shown in italics) relevant to tasks 1622 and 1622a along with the factories for creating those elements.

were not helpful with data-flow issues and where the control-flow was obscured by the use of Java reflection: *"what was throwing me off was how much reflection was being used [...] the hardest part about reflection is that it just breaks the tools"* [1.7/N3]. All of the participants that worked on task 1622 struggled with this issue.

***Observation 6:* Building a complete understanding of the relevant code was difficult.**

As entities and relationships were identified as relevant to the task and more information was discovered about those entities, an integrated model of how those entities fit together needed to be developed: *"I am kind of curious how [the `CoreFactory`] class integrates with this whole hierarchy"* [1.2/N1]. In some cases the tools provided by the development environment acted as an external representation of the information to be integrated [113]. When the information was not or could not be externalized in this way, integration was often unsuccessful even when all (or most) of the relevant pieces of information had been correctly identified. These failures are possibly related to cognitive limitations [30].

For both task 1622 and task 1622a understanding events and how they are created in ArgoUML was crucial. The key classes and interfaces involved are shown in Figure 3.3, though not all of the relationships are shown. Participants generally quickly identified the `MEvent` interface as important and opened it in the type hierarchy which externalized the information shown in the upper left corner of the diagram. The rest of the information necessary to understand ArgoUML events was identified but not externalized as a whole and none of the participants completely understood all of it.

To help build an integrated understanding, some participants drew structural diagrams on paper, presumably to take pressure off of their working memory. To explain why he was writing notes N1 said *"I know I am going to get lost"* [1.2]. During the interview after the session on his use of paper and pen, a participant said: *"I was starting to forget who was calling what, especially because there is only one search panel at a time that I can see"* [1.4/N6]. Even when paper was used, integration remained a difficult and important challenge.

Observation 7: **Programmers' false assumptions about the system were at times left unchecked and made progress on the task difficult.**

An issue that impeded progress in several sessions (particularly 1.9, 1.11 and 1.12), and has not been heavily discussed in the literature, was that of assumptions that were incorrect but never properly checked. These assumptions were only sometimes articulated explicitly. In session 1.9 the root cause of a false assumption appeared to be the misinterpretation of the condition (`!notContained.isEmpty()`) under which a particular exception was thrown: *"the hash table being empty is the problem, right?"* [N8]. This false assumption lead to other false assumptions and significant confusion. The programmers never identified and corrected their root error.

A common assumption was that the existing code in the system was correct. This fueled a desire to reuse system knowledge as observed by Flor *et al.* [18]. An example of this was observed while programmers worked on task 1622 and 1622a during which programmers attempted to use the partial implementation of call events as a guide for implementing other kinds of events. Although this approach was partially successful, the assumption that the existing code was correct was false and caused some problems. The code may have been incorrect because the ArgoUML code base is still in active development.

Observation 8: **Revisiting entities and relationships was common, but not always straightforward.**

Much of the exploration we observed can be viewed as re-exploration. In fact 57% of observed visits to source code entities were revisits and, perhaps unsurprisingly, in many sessions this proportion of revisits increased over time. Several revisiting patterns are depicted in Figure 3.4. In a session such as 1.4, where the one of the participants (N4, in this case) had previously worked on the assigned change task (1622) the proportion of revisits was quite high but generally not increasing, possibly suggesting differences in revisiting behavior at different stages of a task.

In some cases, source code entities appeared to be revisited because the discovered information had been forgotten, and in other cases because an earlier exploration had been stopped short or had been unsuccessful. This revisiting or re-exploration was sometimes intentional and other times not, and was sometimes noticed and sometimes not: *"we were retracing steps we had done before and [were not] aware of it"* [1.2/N1]; *"when we say let's look at that later, I think what that usually means is that if we come across this again using a different route, if there is an intersection somewhere, then we're going to look at this"* [1.3/N4].

A number of different tools were used both for discovering new information and (later) for navigating back to that information. In session 1.5, for example, the programmers used Eclipse's inline type hierarchy both to find out about a type hierarchy and then to navigate between the members of that hierarchy. Similarly the search features of Eclipse were used to initially find a piece of information and then later to navigate back to it.

Some previous research has focused on supporting navigation, including revisiting, (for example, [14]) and the results can be seen in certain features that Eclipse and other state-of-the-practice IDEs have today: back-and-forward and hypertext navigation, for example. Despite these features, the activity of navigating was sometimes simple and direct; other times it was not. In a few rare cases, navigation was really difficult because the programmers

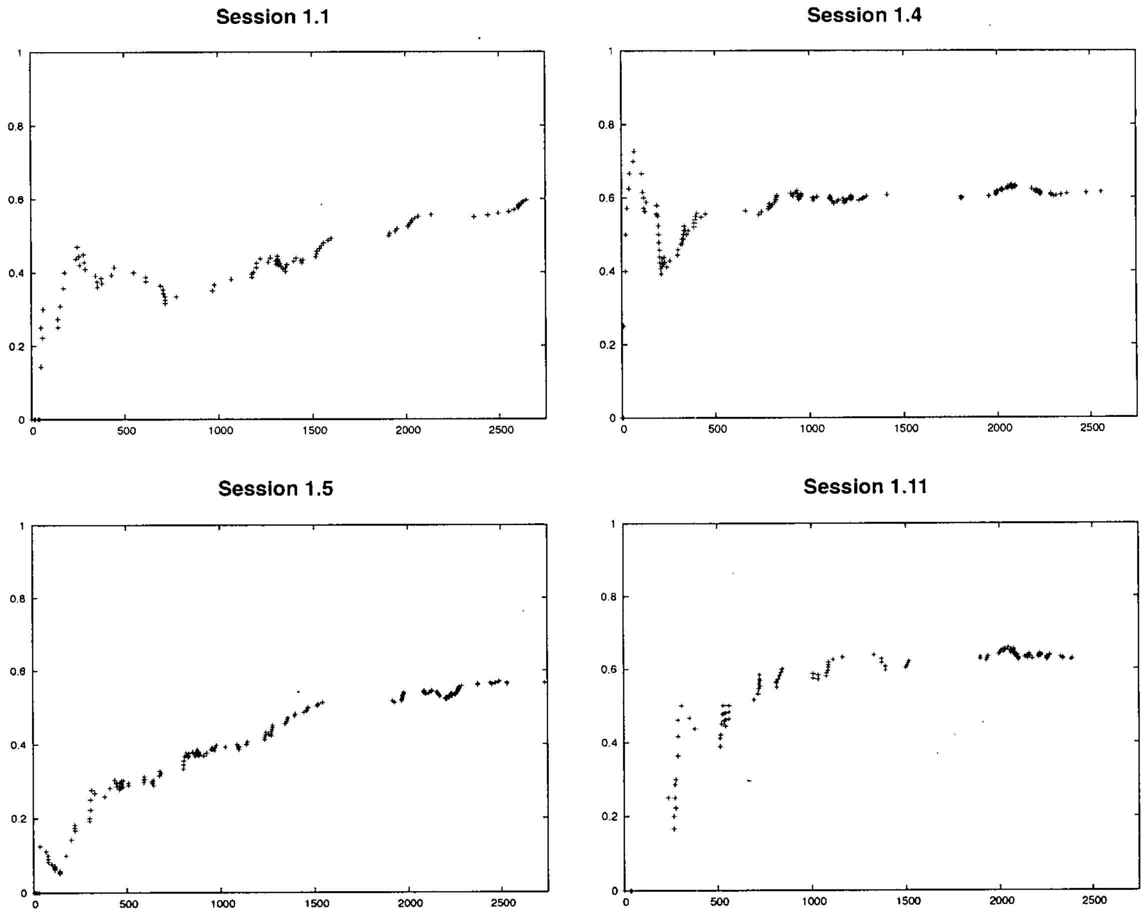


Figure 3.4: Several source code entity revisit patterns from sessions from this study. Time in seconds is along the x axis. Along the y axis is the proportion of visits up to that point in time that are to entities that have been previously visited.

had only a vague handle on the entity they wanted. In these cases, a certain amount of re-exploration was needed to complete the navigation. During session 1.6, the programmers wanted to navigate back to an entity they had visited one minute prior, but had difficulty doing so: *"the class disappeared"* [N5]. Eventually using the editor tab list, after about thirty seconds of effort, they re-found the entity of interest. At one point during session 1.11 the participants completely abandoned the effort to navigate back to a previously visited entity: *"do you remember in that stack trace of the exception where the method was that was sort of doing that save?"* [N2].

3.2.1 Challenges

The participants in our study found the assigned tasks challenging, as we expected they would. Many participants expressed a feeling of having made little progress during a session or specific parts of a session: *"waffled around"* [1.2/N1]; *"we ended up going in circles for a long time"* [1.6/N2]; *"we seemed to be spinning our wheels a bit"* [1.11/N2]; *"I am not sure what we did in terms of progress"* [1.12/N6]. Based on our observations from the previous section, there are at least five important challenges the programmers faced in performing these tasks. Though several of these have been discussed in previous research, our motivation for listing these here is to highlight the challenges that appear to be the most significant to the particular situation we have studied. In Section 5.2.5 we discuss these challenges in the context of our further analysis.

Gaining a sufficiently broad understanding. The participants struggled with gaining a sufficiently broad understanding of the relevant entities and relationships in the code base to correctly complete the task (see Observations 2, 5 and 6). While the observed investigation of the code base began by focusing quite narrowly, as participants identified information of relevance and attempted to build on it, they needed to identify more broad information and form a mental model of more of the code. Participants consistently found this difficult. To the best of our knowledge, this challenge is not directly discussed in the literature, however it does relate to bottom-up comprehension (e.g., [83]) discussed previously (see Section 2.1.1) and the concept assignment problem [4] discussed in Section 6.4, both of which focus on the process of developing a higher-level understanding of a code base.

Cognitive overload. At times the amount of information that needed to be understood exceeded the amount that the programmers could manage mentally, causing cognitive overload [30]. To compensate, the programmers avoided looking at more detail than necessary and tried to minimize the amount of information that they attempted to understand (see Observations 2 and 4). Behaviors such as drawing diagrams and taking notes can be seen as an attempt to bridge local views to generate a more global understanding (see Observation 6) without relying on memory. We are not the first to identify this as a challenge faced by programmers, for example, Storey *et al.* discuss this issue in the context of dealing with multiple programming tools [95].

Navigation. Participants frequently navigated between entities that they had already visited (see Observation 8). Deline *et al.* believe that issues related to navigation and re-finding detract from programmers quickly accomplishing their tasks [15]. We hypothesize that when navigation is not direct, additional cognitive effort is required of programmers, more pressure is put on working memory and developing a complete understanding is made more difficult.

Making and relying on false assumptions. Difficulties around unchecked false assumptions were presented in Observation 7. We believe participants made and failed to check false assumptions for at least three different reasons: (1) they misunderstood information they observed, (2) checking the information was difficult due to tool usage issues (see Observations 1 and 5) and (3) assumptions were not explicitly formulated as hypotheses that needed to be checked. Similar to Robillard *et al.*, we found that relevant information is typically discovered only when deliberately searched for, which we believe allows false assumptions to continue [77].

Ineffective use of tools. Programmers varied in their use of tools. For example, some programmers made effective use of the debugger (see Observation 5) while others did not and as a consequence missed opportunities to directly discover information that they were trying to find. We also observed participants who did not understand the tools provided, such as being unsure of how to interpret search results (Observation 5), and who made less effective use of a particular tool, such as session 1.2 in which the participants could have used the type hierarchy viewer to externalize more relevant information. This challenge appears to be not well studied, however some previous research has investigated the usability of programming tools (e.g., [29]).

3.3 Summary

The first study we carried out was conducted in a laboratory setting. The goal of this study was to observe programmers performing significant change tasks, using state-of-the-practice development tools. The study involved nine participants (with a range of previous experience) and a total of twelve 45 minute sessions. In each session two participants performed an assigned task as a pair working side-by-side at one computer. In the course of this study we observed different pairings working on the same change task, as well as participants working in different pairings. Our data collection has been largely qualitative, with most analysis focusing on an audio recording which was made of discussion between the pair of participants and a video of their screen.

We also covered some initial observations that resulted from our analysis of the data from this study. This discussion included a description of the basic activities we observed, tool usage statistics and eight key observations which are summarized in Table 3.3. The participants in this study found the assigned tasks challenging and in this chapter we have discussed our initial impressions of the major challenges that the participants faced in completing the tasks: *gaining a sufficiently broad understanding* and *ineffective use of tools*, for example. Results from further analysis of the data from this study (along with data from the second study), including further discussion about these challenges, is presented in Chapter 5.

Chapter 4

Study 2: Industry-based Investigation

The second study we performed was conducted in an industry setting. The goal of this study was similar to that of the study described in the previous chapter, that is to observe programmers performing significant change tasks. In this study we observed programmers working on their own change tasks to a code base for which they had responsibility. Sixteen programmers, all employed by the same large technology company, participated in this study. During the sessions participants used whatever tool set they normally used and the systems they worked on were implemented in a number of different programming languages. 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, and the level of prior knowledge of the code base.

This chapter presents the details of this second study (Section 4.1) along with some initial observations of the results (Section 4.2). As described earlier (see Section 1.1) we have used a grounded theory approach in analyzing our data. The observations presented here are based on the categories we developed as part of trying to understand the difference between the situation studied in the laboratory and the situation studied in the industrial setting. These differences include the level of familiarity participants had with the code base, the range of programming languages and tools used, the level of customization in the way tools were arranged, challenges in working in a production environment, and the level of isolation the tasks were performed in. Results from our further analysis of the data from both studies, focusing more closely on the question and answering process, are presented in Chapter 5.

4.1 Study Setup

Our second study was carried out with sixteen programmers in an industrial setting. In this case we studied individual programmers (rather than pairs) because that was how the participants normally worked. Each of the sessions (numbered 2.1 to 2.15) we conducted are summarized in Table 4.1. One session from this study was exceptional in that two participants (E6 and E7) worked together because that was how they were accustomed to working. These two participants referred to their working arrangement as "pair programming", however their work style was different than the pair programming approach used in our first study. E6 and E7 worked on the same task but both used their own computers (laptops), and used NetMeeting to share their desktops and used IM to pass snippets of code to each other.

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. The complete list of programming languages used include: C [49], C++ [97], Java [28], DOS Batch [81], Tcl [19], C# [35], HTML [68], XML [34], ASP [107], SQL [41] and MDX [32]. Each of the tools used are described below.

- Visual Studio: Integrated development environment providing basic static analysis and debugging support. There are versions for a range of 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).

Session	Participant	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	Visual Studio, Enterprise Manager, Query Analyzer, Analysis Manager
2.9	E10	C++, Batch	Notepad, Visual Studio
2.10	E11	C (embedded)	Proprietary loading and debugging tools
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

Table 4.1: Session number, participant(s), the programming language(s) for the system being changes, and the primary tools for each session of the second study.

- NetMeeting: 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, Query Analyzer and Analysis Manager: various tools for administering and creating queries against Microsoft SQL Server.
- Emacs, VI, VIM and UltraEdit: basic source code editors.

Participants were observed working on the their tasks for approximately 30 minutes. After each session the experimenter (the author of this dissertation) 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 of the session.

4.1.1 Participants and Change Tasks

Three of the sixteen participants (E9, E11 and E16) are female. All participants were professional programmers employed by the same large technology company and several of them 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 (or identical in a few cases noted below) code bases. In this study the participants were all working with code that they had experience with, though the amount of experience varied significantly (from a just few months to eight years).

The tasks were selected by the participants. They were asked, in advance of the session, to select a task that would be "involved, not a simple local fix", beyond that no guidance was given. In each session the programmer was asked to describe the task they had selected and then to spend about 30 minutes working on that task. They were asked to think-aloud while working on the task [101]. The following describes the code bases and the tasks the participants worked on as well the amount of experience each programmer had with the code base. The code bases were proprietary and we had no access to them, so these descriptions are based on the comments from the participants. In several cases the participants were not able to specify a size for their code base.

- We observed E1 working on a component comprising 200KLOC to 300KLOC, that is part of a larger system (over 1 million lines of code) which he described as a *"physical design tool"*. He had been working on different aspects of this larger system for eight years. His task was to add a new feature to the component.

- Participant E2 worked on the same component as E1 and his task was to port older code to a newer code base for the component. He also had been working on different aspects of this larger system for eight years.

- We observed participant E3 working on code for a middleware (business transaction processing) application. In particular he was working on code for converting between data types. He had been working on this code base for four years.

- Participant E4 described his task as a refactoring task. The code base he worked on was a number of generic COM components for use in middleware applications. He had been working on that code base for two and a half years.

- Like, participant E3, participant E5 worked on code for a middleware application. His task focused on various generic components that he estimated were a several thousand lines of code each and he had been working with this and similar types of applications for five years.

- Participants E6 and E7 performed a task that involved writing test code for a back-end file transfer application. The code base comprised 20KLOC. Participant E6 had been working on this code for six months. Participant E7 had been working on this code for seven months.

- We observed participant E8 working on a 30KLOC key management application that he had been working on for several months. He felt that he understood this application *"extremely well"* because he had spent several years working with a previous application with the same architecture.

- Participant E9 was observed fixing a bug with a database application. She had spent four years working with a family of such applications.
- The code base E10 worked on is an application that is over a million lines of code, and he described it as a *"large mostly unfamiliar code base"* though he had been working on it for several months. His task was to update the uninstall feature of that application.
- Participant E11's task focused on testing an embedded board management system. She had spent that last 4 or 5 years on this and other embedded systems.
- Participant E12 worked on a restructuring task involving two code bases. The first is an engine (or library) written in C, and the other is a C++ GUI application which used that engine. Just the GUI application comprised 10KLOC. He had been working on these code bases for six months.
- We observed participant E13 performing a relatively simple (he referred to it as *"not that advanced"*) bug fix to a web application that he had been working on for seven months.
- The change task participant E14 worked on involved moving a feature between two versions of a code base. The code base, which he had been working on for two years, comprised over a million LOC, much of it hardware level code.
- Participant E15 worked on the same code base as E14. E15's involved changes to the build system. He had been working on the code base for six months.
- The change task participant E16 worked on was a bug fix to the front-end for a compiler. She had been working (part-time) for three years on this 500KLOC code base.

These sessions have allowed us to observe different programmers working on a wide range of change tasks targeting a range of code bases. This data (along with data from the first study) provides a basis for exploring how programmers manage change tasks.

4.2 Initial Observations

In this section we describe several initial observations based on the data collected from the second study. Our focus is on characterizing several observed differences between the situation studied in the laboratory and the situation studied in the industrial setting. These differences include the level of familiarity participants had with the code base, the range of programming languages and tools used, the level of customization in the way tools were arranged, challenges in working in a production environment, and the level of isolation the tasks were performed in. Each of these initial observations are described below. Results from further analysis of the data from both studies is presented in Chapter 5.

Familiar Code Bases

A central difference between the two studies is that the participants in this second study were observed working on a code base that they had previous experience with, and some participants felt that they knew their code base *"extremely well"* [E8]. As compared to the participants in the previous study, these participants began the task knowing much more about both the application and the application domain. Interestingly, however, several of the participants felt that they did not completely know the code base on which they worked, for example: *"it is as complicated as say the Linux kernel [...] I have only been working on this two years, there is no way I can understand everything, but I understand the stuff that gets assigned to me"* [E14]; *"I haven't been in development mode the last week or so, I am forgetting everything"* [E11]; and *"the project I'm currently working on is a large and mostly unfamiliar code base so I typically need to do analysis of a few days prior to digging into the code"* [E10]. Participant E8 felt that code he wrote during the session needed to be documented because *"when I look at this code in two months I won't remember what I did"*. Also, though participants in this study selected their own tasks and clearly began the session knowing more about the task than participants from the previous study, they did not always immediately know which changes were needed. For example, E10 spent most of the session in what he described as *"figuring out what needs to be done mode"* [E10].

Multiple Languages and Tools

Most of the change tasks we observed participants perform in this study involved systems implemented in multiple programming languages. Of the fifteen sessions in this study (summarized in Table 4.1) only six involved only one programming language (sessions: 2.4, 2.7, 2.10, 2.13 and 2.15). Participants in this study also used multiple tools to interact with their code base. For example, participant E9 used four different tools (Visual Studio and three tools targeting Microsoft SQL Server) to perform her change task and frequently switched between them, which she described as *"not a convenient way to work because we actually refer to the same objects and the same data from different tools"* [E9]. Similarly, participant E11 used multiple tools to support loading, executing and debugging embedded code (*"when you're working through multiple systems like this, you feel like you are jumping through all these little loops"* [E11]). Some participants even considered applications for instant messaging (*"we use IM quite a lot if we have a quick question for someone"* [E4]), email and browsing the web as an important part of the development tool set, in particular when the application was web based. In contrast, participants in our first study worked on a code base written exclusively in Java and they spent the vast majority of their time working with the tools provided by the Eclipse Development Environment.

Customized Environments

Participants in the first study were provided with a computer system to use for the session, and none of those participants customized the provided environment in any significant ways. The participants in the second study were observed working on their own computers, which allowed us to see how they normally arranged their programming tools. Four of these arrangements are depicted in Figure 4.1 and are described below

Participant E1 organized each of his tasks (including the one we observed him working on) around nine virtual desktops: *"I tend to split up everything based on what task I am working on"* [E1]. Part A of Figure 4.1 is a sketch of the way applications were arranged in these virtual desktops. Different desktops, and the applications in those desktops are used for different parts of working on a task: *"the center is where I always keep my Emacs. If I am*

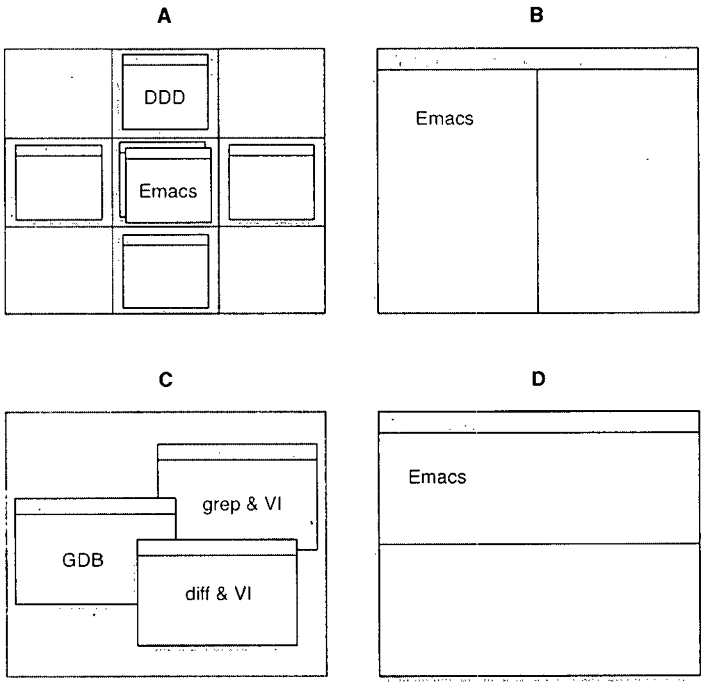


Figure 4.1: Various arrangements of windows and tools observed in study two: (A) E1's use of virtual desktops, (B) E14's split Emacs window, (C) E16's arrangement of shells and tools in those shells, and (D) E2's split Emacs window.

doing any debugging then I go up. To do my merges I go over. You just get used to doing it the same way" [E1]. He also customized various aspects of the individual tools, for example his application windows had no title bars (*"because that is just a waste of real estate"* [E1]).

Participant E14 had a large display and had a maximized Emacs window filling the entire display. The Emacs window was split vertically as sketched in part B of Figure 4.1. The task we observed E14 working on involved porting code from one source tree to another: *"trying to bring over into the code I am working on, but the code is different enough that I can't just do a simple diff, so I got to go into there by hand and find what I need"* [E14]. He explained his reasons for his arrangement by saying *"so I can look at both files, edit both of them without having to click from window to window"* [E14]. Participant E2, who also worked on a merging task, used Emacs arranged as in part D of Figure 4.1.

Participant E16 organized her task around three DOS command windows arranged as shown in part C of Figure 4.1. In one window she primarily used VI to work with the code that was most central to her task. In the second window she searched for and viewed other related source files. The third window was used for running the failing test case in the debugger (GDB). She switched frequently between these three windows: *"I try to always arrange them in the same way so that I can remember where I am at as I windows switch"* [E16]. Though the specifics were different, participant E15 worked in a very similar way.

4.2.1 Production Environments

The participants in the first study worked on an isolated application and knew that the code they wrote would never need to be deployed in a production environment. In contrast, the participants in the second study were often working on code that interacted with a much more complicated environment and they were writing code that would (eventually) be deployed in production and in many cases the applications were crucial to the company. As a result the participants in the second study dealt with a different set of issues: *"when you integrate [a changed] module into the bigger system, something goes wrong"* [E1]; *"you are never going to get 100% coverage from your unit tests, so you really can't know if everything is working the way you want it to work"* [E4]; and *"there's a lot of the interactions between the different modules that aren't exactly understood to be honest, so there's always a chance for a side effect if you modify a piece of code"* [E10].

4.2.2 Interruptions

One final difference we observed between the two situations was that in the first study the participants were isolated and worked exclusively on the assigned change task, while in the second study the participants faced distractions and interruptions as they would in their normal work day. For example, participant E8 was interrupted several times by instant messages: *"this is what sucks about IM: you're in the middle of something and people can get a hold of you"* [E8]. This same participant believed that *"any interruption is about a five minute problem"* and said that he would only program when he had *"at least half an hour"* to dedicate to the activity. Similarly, while we were observing participant E10 working on a change task, he left his cubicle to ask his nearby coworker a question (about what stage of a long-term restructuring the code base was in). Obviously the participants in the first study did not have this opportunity.

4.3 Summary

Our second study was conducted in an industry setting. The goal of this study was similar to that of the first study: to observe programmers performing significant change tasks. In this study we observed programmers working on their own change tasks to a code base for which they had responsibility. During the sessions participants used whatever tool set they normally used and the systems they made changes to were implemented in a number of different programming languages. These two studies, then, 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, and the level of prior knowledge of the code base.

This chapter has presented the details of this second study along with some initial observations of the results. In presenting these observations our focus has been on characterizing several observed differences between the situation studied in the laboratory and the situation studied in the industrial setting. These differences include the level of familiarity participants had with the code base, the range of programming languages and tools used, the level of customization in the way tools were arranged, challenges in working in a production environment, and the level of isolation in which the tasks were performed. Results from further analysis of the data from both studies is presented in the next chapter.

Chapter 5

Questions in Context

This chapter presents the results of our analysis of the data collected in the two user studies described in Chapters 3 and 4. This analysis was an iterative process of discovering questions in the data and exploring similarities, connections and differences among those questions. Our analysis method is described in more detail in Section 1.1. 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. For example, N4 asked the question *"how does [MAssociation] relate to [FigAssociation]?"* which can be made more generic as: *How are these types or objects related?* (see question 22). As this example illustrates, the questions in our catalog are very close to the data. As a result these questions are discussed in Section 5.1 without a comprehensive set of exemplars from the raw data and in other places in this dissertation are used in place of raw data. This use of the questions makes for a more clear presentation and allows us to manage the confidentiality issues around the proprietary code featured in our second study. In cataloging and 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 also report on the results of our analysis of the behavior we observed around answering those questions both at the particular question level and at the category level. In the context of the process of answering questions, we have analyzed the interactions between questions and tools. We found that some questions are closely related (with some questions being asked in support of higher-level questions, for example). We also observed important issues for our participants in using the available tools to answer their questions (with tools answering less precise questions than intended by our participants, for example) and taking multiple disconnected result sets and using those to answer their intended questions. These issues are discussed further in Section 5.2. Further analysis considering a wide range of tools for answering the questions our participants asked is reported on in Chapter 6. Note that the results presented in this chapter have been published previously [84].

5.1 Questions Asked

We report on 44 kinds of questions we observed our participants asking. This catalog captures the information that our participants aimed to discover about the various code bases they worked on. These questions are generalized versions of the specific questions asked by our participants. 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. 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 5.1. 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 (i.e., questions that build on a node). 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).

This categorization, along with the questions we have identified, represent a novel and central contribution of our research, though aspects of programs are often referred to in terms of graphs. For example, Brooks referred to directed graphs representing control flow, data flow and dependencies [7]. Although other categorizations of the questions are possible, we present 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 (a question–subquestion relationship, for example), as well as certain challenges in answering questions (mentally integrating information, for example) as described in Section 5.2.

The levels suggested by our categories bear some resemblance to the hierarchy of information implied by various cognitive models (see Section 2.1.1), 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, for example, a bottom-up (e.g., [83]) or a top-down (e.g., [88]) model. 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]).

5.1.1 Finding 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]. 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 starting point. These are perhaps similar to what Wilde and Casey call "places to start looking" [109].

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*

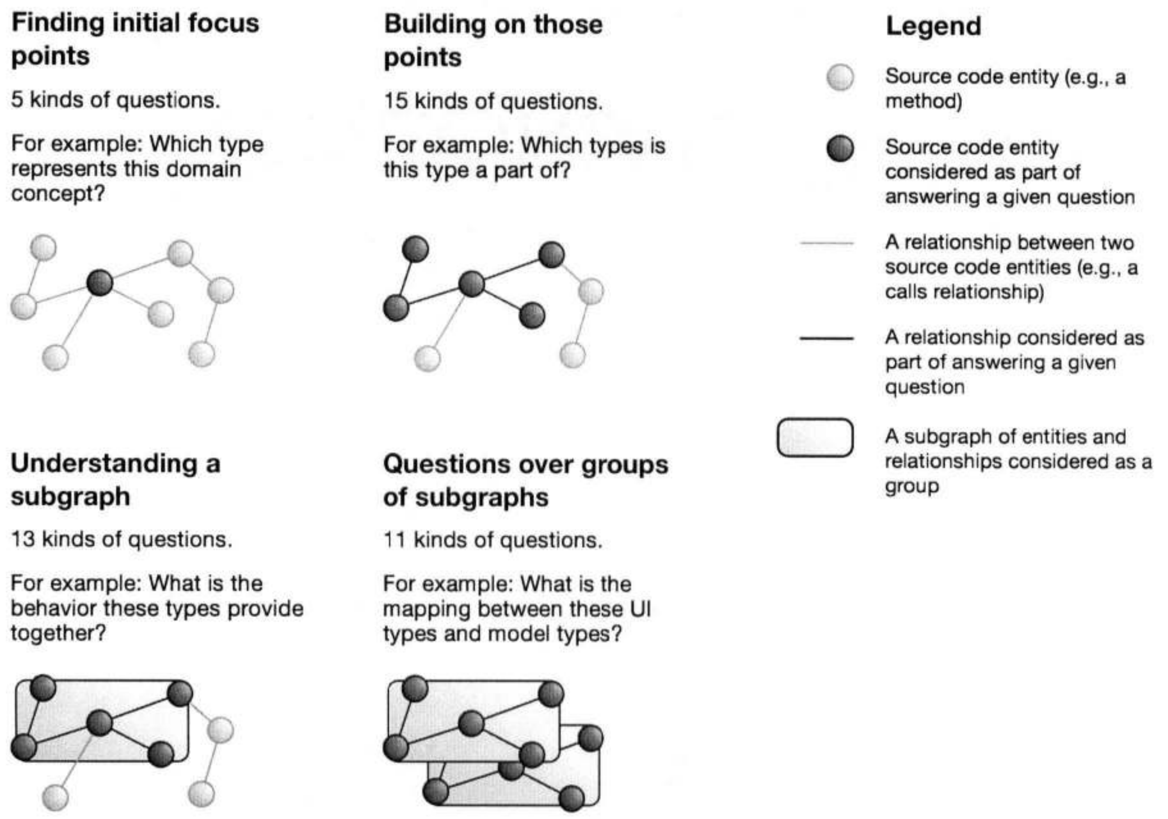

Finding initial focus points
5 kinds of questions.
For example: Which type represents this domain concept?
Building on those points
15 kinds of questions.
For example: Which types is this type a part of?
Understanding a subgraph
13 kinds of questions.
For example: What is the behavior these types provide together?
Questions over groups of subgraphs
11 kinds of questions.
For example: What is the mapping between these UI types and model types?
Legend
Source code entity (e.g., a method)
Source code entity considered as part of answering a given question
A relationship between two source code entities (e.g., a calls relationship)
A relationship considered as part of answering a given question
A subgraph of entities and relationships considered as a group

Figure 5.1: 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.

UML editors]" [N5]; and *"my idea is to see if we can find a representation of this transition"* [N1]. 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]; and *"do a search for spline"* [N3] (where spline was the text in a tool tip). 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)
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 (project, package or class, say)? (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 was used (for example, Eclipse's Package Explorer, which provides a tree view of the packages and classes in a system). 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].

5.1.2 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.

Sometimes we observed a series of questions about the same entity, forming a star pattern as depicted in part A of Figure 5.2 (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 5.2.

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)

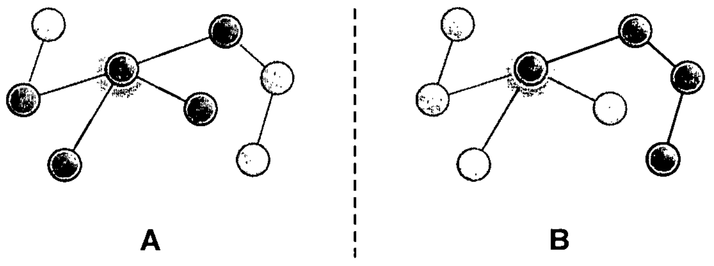


Figure 5.2: A depiction of two observed patterns of questions: (A) multiple questions about the same entity, and (B) a series of questions where each subsequent question is about a newly discovered entity. Circles represent source code entities and lines represent relationships between entities. Black circles and lines represent the the entities and relationships visited as part of answering a series of questions.

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]; *"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)

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].

5.1.3 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, 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 5.1.1 and 5.1.2; *"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?"* [N2] and *"why is the namespace null?"* [N2].

5.1.4 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 5.1.3 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]. 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 is 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]; *"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 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 1.10 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: *"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)
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)

5.2 Answering Questions

To this point we have described specific kinds of questions asked by our participants organized around the four categories summarized in Figure 5.1. 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 5.3. Specifically, we found three important

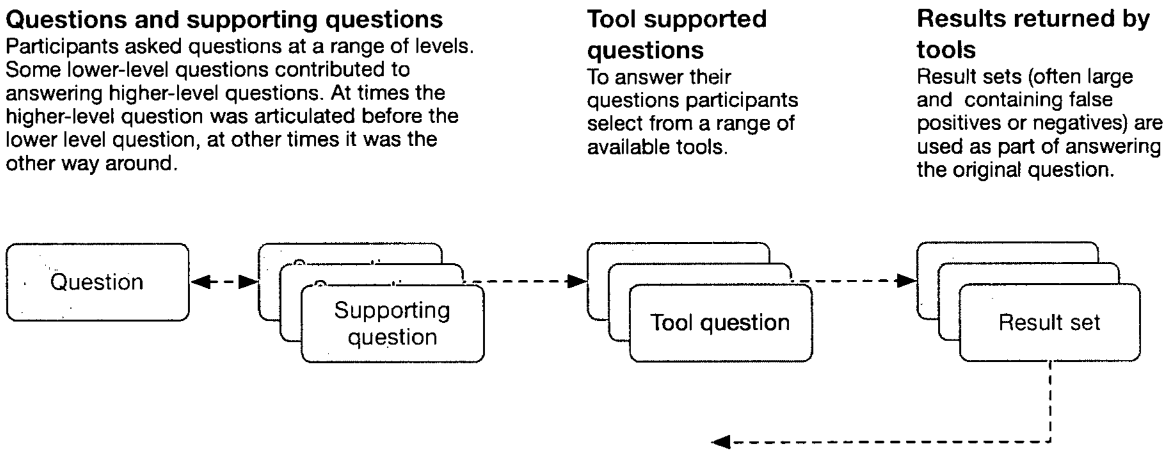

Questions and supporting questions
Participants asked questions at a range of levels. Some lower-level questions contributed to answering higher-level questions. At times the higher-level question was articulated before the lower level question, at other times it was the other way around.
Tool supported questions
To answer their questions participants select from a range of available tools.
Results returned by tools
Result sets (often large and containing false positives or negatives) are used as part of answering the original question.
Question
Supporting question
Tool question
Result set

Figure 5.3: An illustration of the process of using the tools available to answer questions about a software system, showing the relationship between questions, sub-questions, tool questions and results from tools.

considerations. First, the interaction between a participant's questions, with some questions being asked as part of answering other questions (see Section 5.2.1). Second, the interaction between the questions our participants asked and those that the available tools could answer (see Section 5.2.2). Finally, taking the results produced by the tools and using those to answer a participant's intended question (see Section 5.2.3). We present a more detailed anecdote from the second study that provides an overview of this process and many of the issues raised (see Section 5.2.4). We end this discussion by considering some of the challenges we observed in the first study in the context of the results presented here (see Section 5.2.5).

5.2.1 Questions and Sub-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. 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?*). 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. The process of breaking questions down into lower-level questions that were directly supported (or at least 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].

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 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.

We also 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?*). Generally the less refined versions specify less of the context important to the question.

5.2.2 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 process 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, participant 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 approach 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]. Participant E14's experience was similar: *"the code is different enough that I can't just do a simple diff, so I got to go into there by hand and find what I need, extract it out and put in where I need it in my tree"* [E14].

A programmer's questions often have an explicit or implicit context or scope. In question 31 (*Which execution path is being taken* <u>*in this case*</u>*?*) this context is explicit. Question 33 (*What parts of this data structure are accessed in this code?*) asks about changes to a data structure but in the context of a certain section of code. For example, in session 1.6 the participants used the call stack in the debugger to very manually try to understand the relationship between a given data structure and a sequence of method calls. As another example, in session 1.11, participant N2 wanted to learn about the properties of an object in the context of a particular failing case of a large loop: *"what I want to see is the object that is being converted to XMI"* [N2]. Getting a handle on this object (in the debugger, for instance) proved difficult. We observed that the tools used by our participants had very limited support for specifying the scope of a tool or query with the result that information displayed was noisy: *"we need to narrow this down"* [N4].

5.2.3 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 consequence holds 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].

5.2.4 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 often 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 results, 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 participant E16 worked on was a connected group of data structures which she described as *"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 5.1.3).

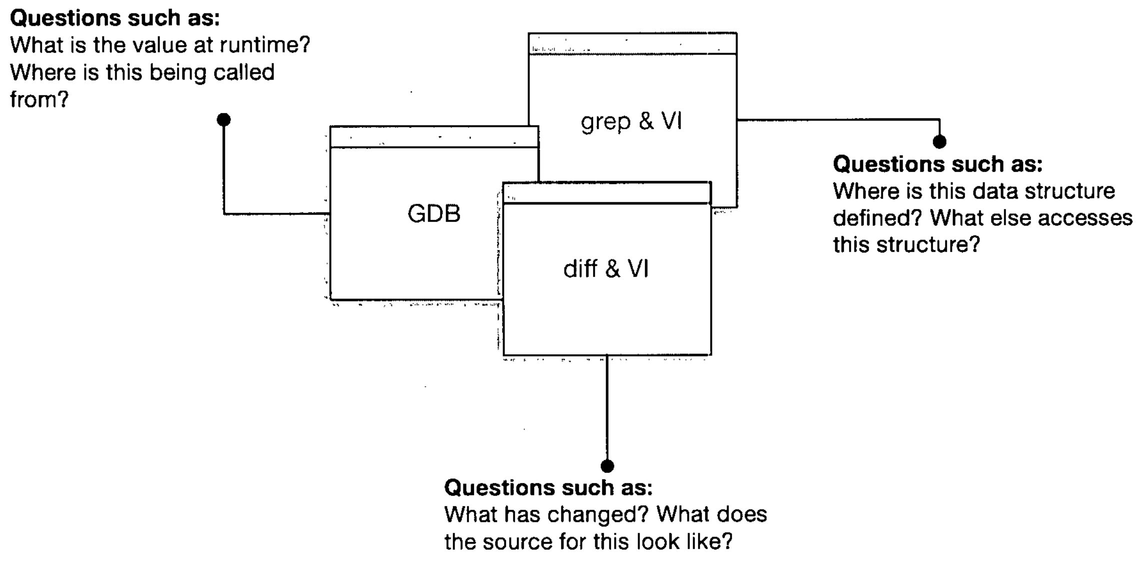

Questions such as:
What is the value at runtime? Where is this being called from?
grep & VI
GDB
diff & VI
Questions such as:
Where is this data structure defined? What else accesses this structure?
Questions such as:
What has changed? What does the source for this look like?

Figure 5.4: Participant E16's arrangement of windows and associated tools. Also shows the types of questions asked in each window.

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 5.4. 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"*.

5.2.5 Understanding the Challenges Programmers Face

At the end of Chapter 3 we presented five significant challenges faced by the participants of study one: (1) *gaining a sufficiently broad understanding*, (2) *cognitive overload*, (3) *navigation*, (4) *making and relying on false assumptions*, and (5) *ineffective use of tools.* Here we discuss these challenges further in the context of the results we have presented in this chapter.

- Cognitive overload and issues around gaining a sufficiently broad understanding of the portions of the source code relevant to the task are particularly relevant to questions that require considering one or more subgraphs of information. In general learning *Which types is this type a part of?* is easier than developing an understanding of how a particular type *"fits into GUI generation as a whole?"* [N3], for example.

- Issues around navigating relate to the need to use multiple tools to answer questions and needing to piece together answers. We expect that being able to do this accurately would facilitate answering higher-level questions as it would be easier to mentally maintain the necessary context.

- Making and relying on false assumptions becomes more likely if checking those assumptions is difficult (i.e., when questions can not be answered directly using the available tools). When the information presented in tools is noisy (due to the programmer not being able to appropriately specify a scope for a question, for example) information is more likely to be missed and false assumptions not corrected.

- For a given question, various tools could potentially be used to answer those questions, with some being more effective than others. We observed that effective use of tools could significantly improve the success a participant had in answering their questions, however the choice of tool is not always straightforward as there is often no perfect match.

5.3 Summary

We have considered the questions our participants asked at several different levels. First, we have produced generic versions of the questions which has allowed as to abstract slightly from the specific situations and code bases. These abstractions have given us the ability to compare these questions and record their frequency across sessions. For example, question 4 (*Is there a precedent or exemplar for this?*) was asked in six different sessions, while question 20 (*What data is being modified in this code?*) was asked in just two different sessions.

Next, based on these generic questions we have developed four categories and several subcategories which are based on the information needed for answering those questions: (1) questions about finding initial focus points, (2) questions about building on such points, (3) questions aimed at understanding a subgraph, and (4) questions over multiple subgraphs.

Finally we have considered the overall process of using tools to answer questions, including the relationships between questions. This answering process and the specific questions asked is heavily influenced by the tools available. Many questions are asked only because the intended questions can not be asked directly. A question can go beyond what tools can answer by the scope of information needed to answer it, or a question can be more precise than can be expressed in the tools. Even for lower-level questions asked by programmers the match between the intended questions and the questions the tools were designed to answer was not always perfect. The consequence of these issues is that programmers are presented with noisy information and must mentally piece information together from multiple tools or views.

Chapter 6

Analysis of Tool Support for Answering Questions

In the previous chapter we presented the results of our analysis in two major parts. First, we analyzed the questions asked by our participants which produced a list of generic questions organized into four top-level categories. Second, we analyzed the process of asking and answering questions. This analysis was based on the two studies that we have performed and which cover a particular set of programming tools (Eclipse, Emacs and Visual Studio, for example). In this chapter we build on this analysis by considering the ability of a much wider range of tools (both industry and research tools) and techniques to support a programmer in answering each of the questions we observed as well as how well the overall process of answering questions is supported.

To do this we have reviewed the literature on programming tools and techniques for exploring source code information, and also drew on the tools we observed our participants using. We have performed this review and evaluated the level of support provided, in the context of our participants' behavior around answering their questions. Our goal has not been to find all tools or techniques applicable to each questions, but rather to determine whether or not a tool or technique exists to address each question. In particular we were interested in a tool that fully supports answering each question. Where we have not found such a tool, and also in the absence of evidence from our studies on how a particular industry tool supported our participants, or supporting empirical evidence from the literature, we discuss various candidate tools that may provide some support. In this way we develop an understanding of where existing industry tools work well, where researchers have proposed relevant techniques, and where support is currently lacking. This understanding is important for identifying areas where further research or new tools would be valuable.

This discussion is organized around the four categories of questions. Tables 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 summarize the techniques and tools applicable to answering each question and the level of support provided by those techniques and tools. We rate the level of support provided as *full* (i.e., well supported), *partial* (i.e., some support but not complete), and *minimal* (i.e., little or no tool support). To be clear, our intention is to explore the support provided by tools and techniques, rather than to provide a conceptual framework for categorizing tools or techniques (for an example of such a framework, see [100]). The last section of this chapter summarizes the gap our analysis has exposed between the support programming tools provide and the support programmers need to answer their questions. For example, we show that programmers need better support for maintaining context and for putting information together.

6.1 Answering Questions Around Finding Initial Focus Points

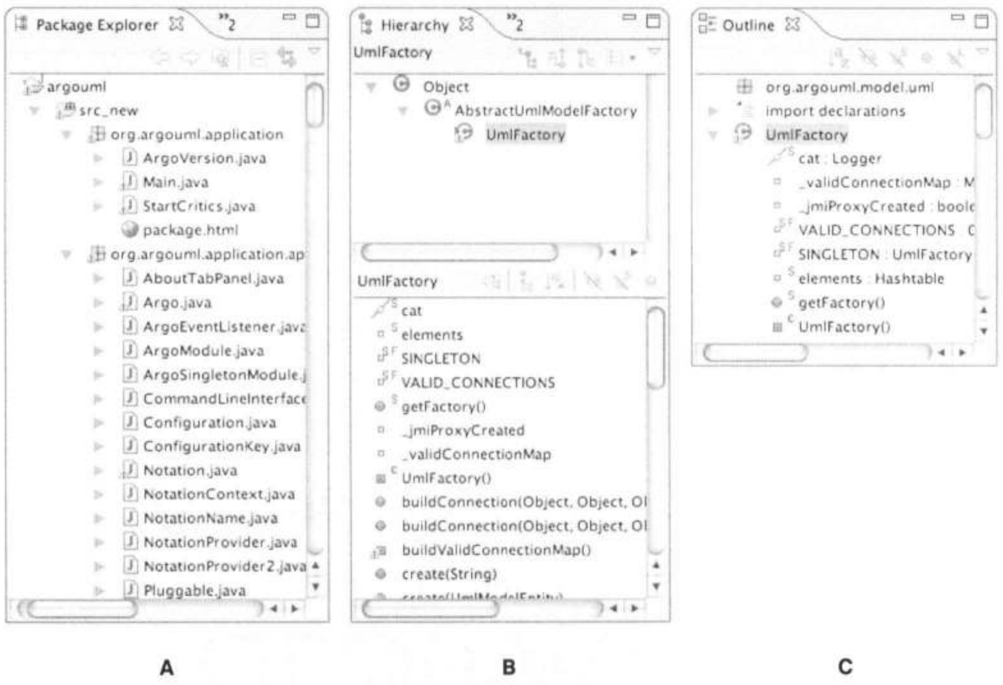
In this section we discuss tool support for answering questions 1 through 5, that is questions around finding an initial focus point in the code base to support further exploration. These questions were particularly important for the participants in our first study at the beginning of a session or when they began to explore a new part of the system on which they were working. Table 6.1 lists some of the techniques and tools applicable to each of these questions, as well as the level of support provided by those techniques.

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 and the Eclipse file search tool can be helpful in answering these questions in many situations. For instance a developer using 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. The Eclipse Open Type tool provides more focused support for this particular scenario as it uses static analysis to limit the search to type names.

Question 5 (*Is there an entity named something like this in that unit (project, package or class, say)?*) can also be supported by source code editors and overview tools such as those shown in Figure 6.1. These are particularly helpful in situations where a hypothesis about the name to locate is difficult to form. This difficulty was encountered, for example, in session 1.6 where the participants were looking *"for a move or a set destination or something like that"* [N2] and resorted to scrolling/scanning through the source code in an editor and then using Eclipse's Outline View (see part C of Figure 6.1) to try to find something appropriate.

Answering question 3 (*Where is there any code involved in the implementation of this behavior?*) was essentially about finding any point in the code relevant to a particular portion of a task. For several of our participants this involved first using lexical or static analysis based tools to generate candidate types or methods. They then used further exploration or debugging techniques to check the hypothesis that a given candidate was part of the behavior of interest (*"get confidence in my hypothesis"* [N6]). Examples of search tools used in this way in our studies include grep, Eclipse's search tools (Figure 6.2 shows Eclipse's Search Results View) and Visual Studio's search tools. Examples of debugging tools used in our two studies include GDB, DDD and the Eclipse Debug perspective.

This multistep process was not always straightforward and at times included several failed attempts. Software reconnaissance is a technique based on comparison of traces of different test cases, which can allow programmers to more directly find a relevant point in the code [109]. This technique relies on a relatively comprehensive test suite and has three parts. First, the target program must be instrumented so that a trace is produced of the blocks or branches executed in each test. Then test cases are run, some "with" and others "without" the desired behavior. Finally, the traces are analyzed to look for blocks or decisions that are executed exclusively "with" the feature and a list of candidate entities is produced which can then be explored further. RECON2 and RECON3 are examples of tools providing support for this technique [42].

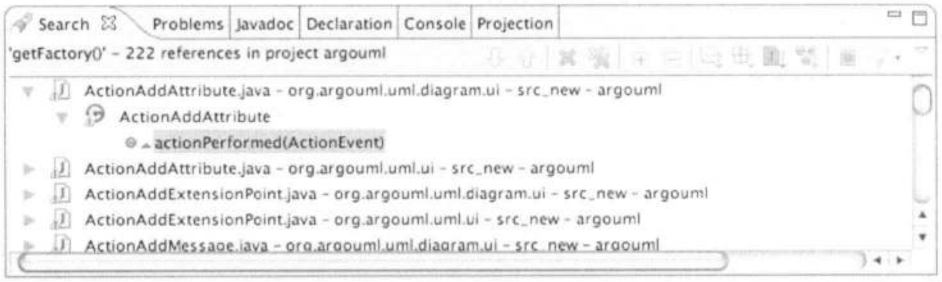


```
argouml
  src_new
    org.argouml.application
      ArgoVersion.java
      Main.java
      StartCritics.java
      package.html
    org.argouml.application.ap
      AboutTabPanel.java
      Argo.java
      ArgoEventListener.jav
      ArgoModule.java
      ArgoSingletonModule.j
      CommandLineInterfac
      Configuration.java
      ConfigurationKey.java
      Notation.java
      NotationContext.java
      NotationName.java
      NotationProvider.java
      NotationProvider2.java
      Pluggable.java
```

```
UmlFactory
  Object
    AbstractUmlModelFactory
      UmlFactory

UmlFactory
  cat
  elements
  SINGLETON
  VALID_CONNECTIONS
  getFactory()
  _jmiProxyCreated
  _validConnectionMap
  UmlFactory()
  buildConnection(Object, Object, Ol
  buildConnection(Object, Object, Ol
  buildValidConnectionMap()
  create(String)
```

```
org.argouml.model.uml
import declarations
UmlFactory
  cat : Logger
  _validConnectionMap : M
  _jmiProxyCreated : boolc
  VALID_CONNECTIONS : C
  SINGLETON : UmlFactory
  elements : Hashtable
  getFactory()
  UmlFactory()
```

Figure 6.1: Three tools provided as part of the Eclipse Java Development Environment: (A) the Package Explorer which shows the package, file and class structure of a code base, (B) the Type Hierarchy which shows portions of a code base's inheritance structure, and (C) the Outline View which provides an overview of the contents of a particular source file.



```
'getFactory()' - 222 references in project argouml
  ActionAddAttribute.java - org.argouml.uml.diagram.ui - src_new - argouml
    ActionAddAttribute
      actionPerformed(ActionEvent)
  ActionAddAttribute.java - org.argouml.uml.ui - src_new - argouml
  ActionAddExtensionPoint.java - org.argouml.uml.diagram.ui - src_new - argouml
  ActionAddExtensionPoint.java - org.argouml.uml.ui - src_new - argouml
  ActionAddMessage.java - org.argouml.uml.diagram.ui - src_new - argouml
```

Figure 6.2: The Eclipse Search Results View showing all 222 references to a method named getFactory.

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 [21])
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 [42])
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 [36])
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 or overviews

Table 6.1: A summary of the techniques and tools applicable to answering each of the questions from the first category, along with the level of support for each: full, partial or minimal. All of these questions are about finding a starting point to begin further investigation and are described in detail in Section 5.1.1.

Answering question 4 (*Is there a precedent or exemplar for this?*) requires identifying a location in a code base that gives information about how to code certain types of things in the context of that code base. Lexical search or static analysis based cross-referencing tools can be used to help (i.e., tools that allow programmers to elicit relationships between source code entities [98, 90]). 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 (Figure 6.2 shows the results of a references search in Eclipse) to identify an example. 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 [36] and Strathcona [40]. CodeFinder supports a programmer in defining queries to be used to find examples in an example-based programming environment (i.e., a repository of tagged examples). These queries are iteratively formed, beginning with a simple textual query which when executed returns a list of possible terms. In Strathcona queries are created automatically and are based on structural context; as a programmer writes code, supporting examples can be (based on the code the programmer has written) automatically proposed.

In summary, answering these questions generally involves performing searches based on 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 relatively 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 by various tools, much of which is irrelevant to the intended question.

6	**What are the parts of this type?**
	Partial support: Source code editors or overviews
7	**Which types is this type a part of?**
	Full support: Static analysis based cross-referencing tools (e.g., Masterscope [98])
8	**Where does this type fit in the type hierarchy?**
	Full support: Static analysis based 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
10	**Where is this field declared in the type hierarchy?**
	Full support: Static analysis based type hierarchy tools
11	**Who implements this interface or these abstract methods?**
	Full support: Static analysis based type hierarchy tools

Table 6.2: A summary of the techniques and tools applicable to answering questions 6 to 11, along with the level of support for each: full, partial or minimal. All of these questions are from category two and are about types and static structure.

6.2 Answering Questions Around Building on a Point

In this section we discuss tool support for answering questions 6 through 20. These are all questions around building on a given point in the source code. With these questions programmers can learn more about a given entity (determine its relevance, for example) and find more information relevant to the task. Answering these questions generally involves considering information about different types of relationships between the starting point and other related entities. Tables 6.2, 6.3 and 6.4 list some of the techniques and tools applicable to each of these questions, as well as the level of support provided by those techniques.

Questions 6 (*What are the parts of this type?*), 16 (*What data can we access from this object?*) and 20 (*What data is being modified in this code?*) are perhaps three of the least well supported questions in this category. Questions 6 and 16 both require considering the

12	**Where is this method called or type referenced?**
	Full support: Static analysis based cross-referencing tools (e.g., CScope [90])
13	**When during the execution is this method called?**
	Full support: Debugging tools (e.g., GDB [22])
14	**Where are instances of this class created?**
	Full support: Static analysis based cross-referencing tools
15	**Where is this variable or data structure being accessed?**
	Full support: Static analysis based cross-referencing tools or slicing techniques
16	**What data can we access from this object?**
	Partial support: Source code editors or overviews

Table 6.3: A summary of the techniques and tools applicable to answering questions 12 to 16, along with the level of support for each: full, partial or minimal. All of these questions are from category two and are about incoming connections.

methods and/or fields of a given type. Source code editors or overview tools can help with answering these. For participants in the first study, Eclipse's Outline View helped, by providing 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 (and task relevant) structure to answer these questions was not straightforward due to the volume of information presented by the various views. Question 20 requires considering a block of code and determining its effects (or the subset of those effects most relevant). We observed participants carefully reading source code to answer this question. Baniassad and Murphy [2], and Jackson [44] 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 analysis. Examples of such tools include cross-referencing tools such as those mentioned above as well as FEAT [78] and JQuery [46], two Eclipse plugins that both add direct support for answering question 14. Note however that we observed that these tools are less helpful when Java reflection or other indirection obscured the control-flow. Also, we 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 or chopping techniques may help [106, 33, 45]. Both slicing and chopping are data-flow analysis techniques that can be used to identify code that potentially impacts the value of a variable at a specified point in the code.

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 similarly be 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, and we observed that this feature was used frequently by our participants. Several participants (E1, E2 and E14) in the second study used a tool called ctags to support this kind of navigation [39]. Ctags uses lexical analysis to produce (as part of a code base's build process, generally) an index file for the source code which can be used by various editors (Emacs in the cases we observed) to navigate directly to the declaration of a source code entity.

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 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 (see part B of Figure 6.1) and was one of the more frequently used tools by participants in the first study (accounting for 14.6% of the events). Eclipse also supports searches for implementing methods.

17	**What does the declaration or definition of this look like?**
	Full support: Lexical (or static analysis) based cross-referencing tools (e.g., ctags [39])
18	**What are the arguments to this function?**
	Full support: Lexical (or static analysis) based cross-referencing tools
19	**What are the values of these arguments at runtime?**
	Full support: Debugging tools (e.g., GDB [22])
20	**What data is being modified in this code?**
	Partial support: Source code editor or data-flow analysis techniques

Table 6.4: A summary of the techniques and tools applicable to answering questions 17 to 20, along with the level of support for each: full, partial or minimal. All of these questions are from category two and are about outgoing connections.

Questions 13 (*When during the execution is this method called?*) and 19 (*What are the values of these arguments at runtime?*) consider 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, and to view the call stack (to answer question 13) and the values of variables (to answer question 19) at that point. 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 the starting point and other related entities. In most cases, eliciting this information is relatively well supported by static analysis based tools such as have been available for many years (e.g., see [27, 70]). For other cases debugging tools, overview tools or data-flow techniques provide some support. Where there are 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 Tables 6.2, 6.3 and 6.4.

6.3 Answering Questions Around Understanding a Subgraph

In this section we discuss tool support for answering questions 21 through 33. These questions are about building an understanding of concepts in the code that involved multiple relationships and entities. Answering these questions requires the right details as well as an understanding of the overall structure of the relevant subgraph. We consider a range of research tools that provide partial support for answering these questions, listing some of the applicable techniques and tools in Tables 6.5 and 6.6.

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 which we described in Section 6.1. Another technique is known as the dependency graph method [13]. This technique involves a search in the component dependency graph, beginning at a starting node. In each step one component (generally a function) is selected for a visit. The programmer explores the source code dependency graph to understand the component and decide if it is relevant to the feature. The search then proceeds to another component until all the components related to the given feature are found and understood. In a case study comparing software reconnaissance with the dependency graph method, Wilde *et al.* found that while software reconnaissance is effective at identifying code associated with a feature, it is less effective at helping programmers in understanding a feature because insufficient context is provided [108]. On the other hand the dependency graph method is very manual (essentially a more systematic version of the behavior we observed in our studies) and involves asking a series of lower level questions to produce information toward answering a higher-level question.

Answering questions 30 (*Why isn't 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 or data flow in a particular context. An *interrogative debugging* tool by Ko and Myers called Whyline aims to help programmers both ask and answer these kinds of questions [52]. This tool uses program slicing techniques (as described above) to present the programmer with an automatically populated question menu organized into "why did" and "why didn't" questions. Under each of these are submenus organized around the objects that could have been affected. These then contain the questions that can be directly answered. The whyline presents the answer to a questions in terms of a visualization of the control and data flow. Whyline has been evaluated and found to significantly decrease debugging time for programmers using the Alice programming environment. Scaling this tool or approach to the level of systems and tasks we observed in our studies remains an open research problem.

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?*), all require considering a range of information, and we observed that considering dynamic information about data and about control flow can aid in answering these and our participants made frequent use of the debugger (GDB or the Eclipse Debug perspective) in this effort. There have been several efforts to support the visualization of data structures at runtime [73]. For example, the Amethyst (later called Pascal Genie) debugging tool provides a visualization of the data (including records, arrays and pointers) currently on the stack [69]. Similarly there have been efforts to provide visualizations to support the understanding of control flow. Some of this work has been part of efforts to teach algorithms to students. For example, BALSA (later called Zeus) animates algorithms including providing a display of the call stack [10, 11]. Though at times more graphical, these tools operate at a similar level as the various tools provided by the Eclipse Debug perspective. The UWPI tool is an example of an effort to provide more abstract information [37], though this effort (like others) is quite application (i.e., algorithm) specific ("fine-tuned it until it worked for a particular example" [72, page 227]), which makes such tools less useful in the context of performing a change task, though there has been some work to minimize the effort to create

21	**How are instances of these types created and assembled?**
	Minimal support: Visualization or browsing tools (e.g., SHriMP [94])
22	**How are these types or objects related? (whole-part)**
	Minimal support: Visualization or browsing tools
23	**How is this feature or concern (object ownership, UI control, etc.) implemented?**
	Partial support: Feature location techniques (e.g., software reconnaissance [109])
24	**What in this structure distinguishes these cases?**
	Minimal support: Visualization or browsing tools (e.g., FEAT [78])
25	**What is the behavior these types provide together and how is it distributed over the types?**
	Minimal support: Visualization or browsing tools (e.g., Relo [87])
26	**What is the "correct" way to use or access this data structure?**
	Minimal support: Visualization or browsing tools
27	**How does this data structure look at runtime?**
	Minimal support: Debugging and data structure visualization tools tools (e.g., Amethyst [69])

Table 6.5: A summary of the techniques and tools applicable to answering questions 21 to 27, along with the level of support for each: full, partial or minimal. All of these questions are from the third category and are about behavior and structure.

a visualization [64]. The challenge for programmers is similar both when using these research tools and when using today's debuggers: what they need to understand goes beyond what these tools present and answering the questions requires a number of investigations around lower level questions.

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 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 bringing together a range of static and/or 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 have been developed which may make the navigation or revisiting aspect of this process more direct, and thereby may make these questions easier to answer. Examples include Lemma which supports navigating through various code paths [60], FEAT which allows a developer to navigate between entities designated as part of a named concern [78] and JQuery which presents the results from multiple queries as a tree that supports navigating between entities in the results [46]. Activities around bringing together a number of relevant entities and navigating between those may provide some minimal support for answering questions such as 22, 24 and 25.

The goal of various software visualization tools is to support a programmer in exploring and understanding programs. We believe that a visualization with the right entities and relationships may provide some support for answering the questions in this category, so we describe two example tools: the Simple Hierarchical Multi-Perspective (SHriMP) tool [94] and Relo [87]. SHriMP is designed to support software exploration, and supports multiple layouts of source code structural information including: grid, spring, radial, tree, and treemap. The tree layout is shown in part A of Figure 6.3. SHriMP provides various kinds of abstraction mechanisms generally aimed at exploring code based on its hierarchal structure, but not specific support for producing a view of the information required for answering questions

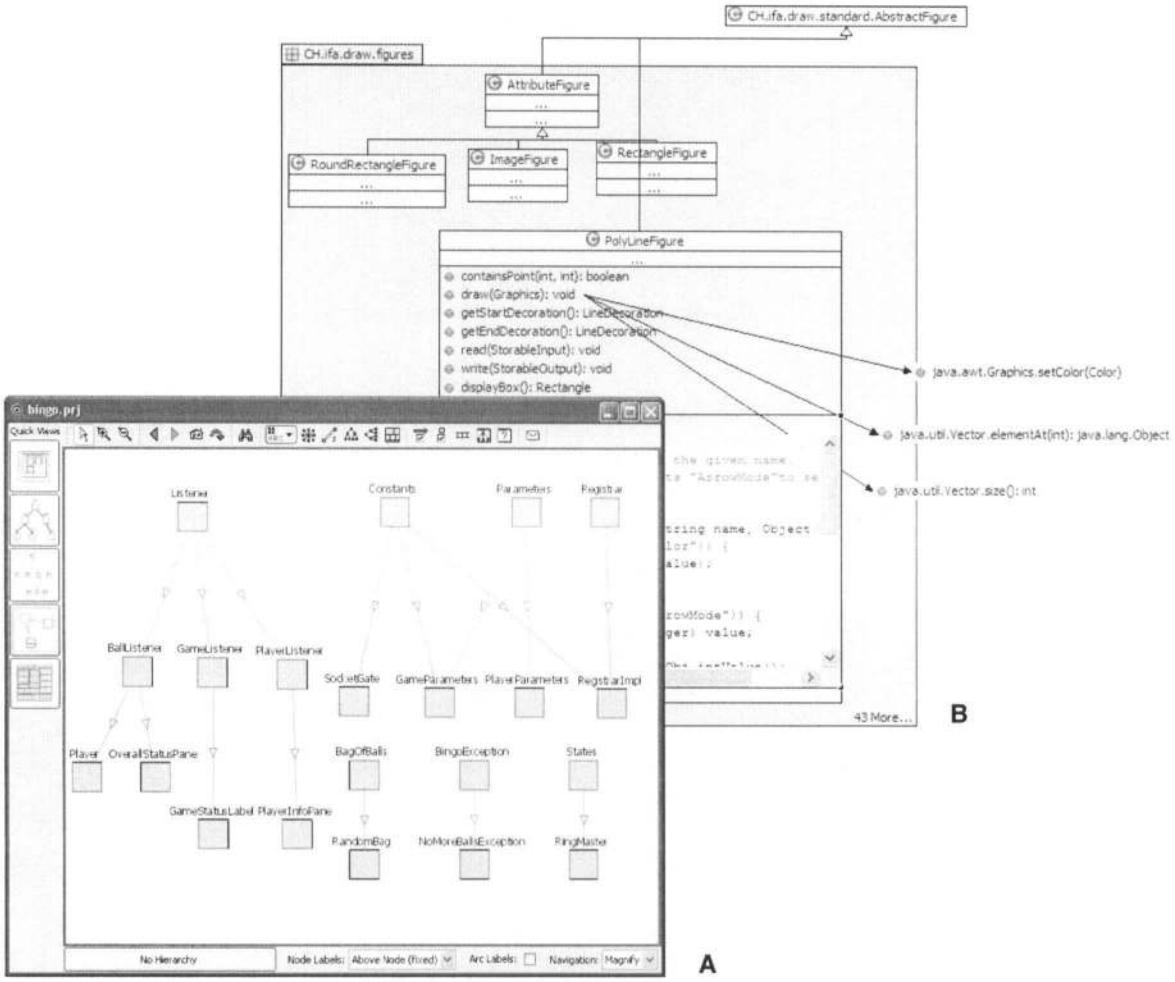


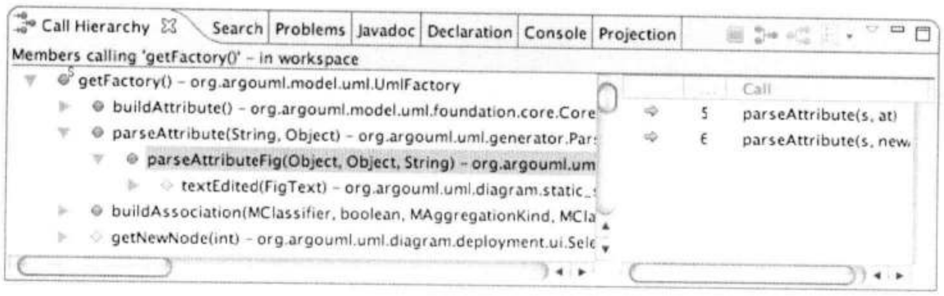
Figure 6.3: (A) A SHriMP tree layout showing part of a system's type hierarchy. (B) A Relo visualization showing a number of classes in one package (CH.ifa.draw.figures), along with source code details for one method (setAttribute) and several connections to methods and types outside of that package.

28	**How can data be passed to (or accessed at) this point in the code?**
	Minimal support: Runtime visualization tools (e.g., BALSA [10])
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 [52])
31	**Which execution path is being taken in this case?**
	Partial support: Debugging and slicing techniques
32	**Under what circumstances is this method called or exception thrown?**
	Minimal support: Debugging and visualization tools
33	**What parts of this data structure are accessed in this code?**
	Minimal support: Browsing or overview tools

Table 6.6: A summary of the techniques and tools applicable to answering questions 28 to 33, along with the level of support for each: full, partial or minimal. All of these questions are from the third category and are about data and control flow.

such as 21 (*How are instances of these types created and assembled?*) and 34 (*How does the system behavior vary over these types or cases?*).

SHriMP (along with Rigi [65] and SNIFF+ [50]) was featured in an experiment which aimed to determine how tools affect how programmers understand programs [95]. The experiment showed that the amount of information (especially arcs) presented could be overwhelming, even for the relatively small program used (1700 LOC). The participants in our studies faced this challenge as well; being able to clearly see and work with the relevant subgraph(s) of the system was difficult. Relo is a tool that aims to support program understanding by allowing interactive exploration of code. As a programmer explores relationships found in the code, Relo builds a visualization (which is initially empty) of what has been explored preserving one kind of context. An example of this is shown in part B of Figure 6.3. Though the process still revolves around lower-level questions and



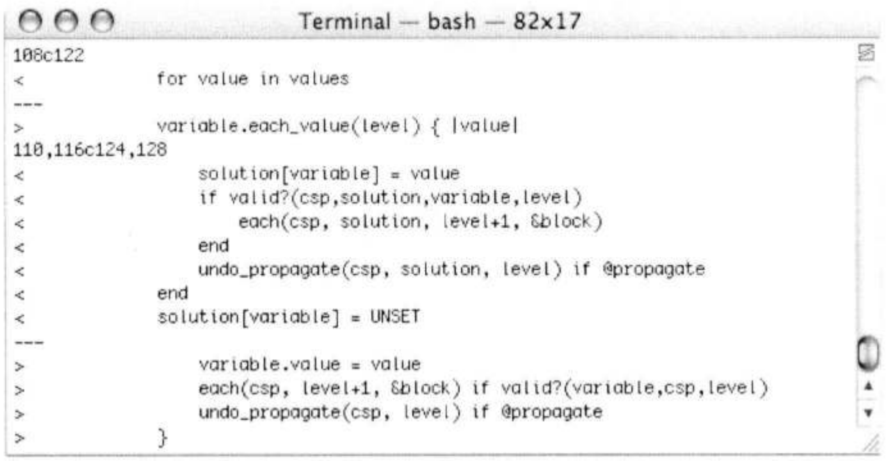
```
Members calling 'getFactory()' - in workspace
getFactory() - org.argouml.model.uml.UmlFactory
    buildAttribute() - org.argouml.model.uml.foundation.core.Core
    parseAttribute(String, Object) - org.argouml.uml.generator.Par:
        parseAttributeFig(Object, Object, String) - org.argouml.um
            textEdited(FigText) - org.argouml.uml.diagram.static_:
    buildAssociation(MClassifier, boolean, MAggregationKind, MCla
    getNewNode(int) - org.argouml.uml.diagram.deployment.ui.Selc

        Call
    5   parseAttribute(s, at)
    6   parseAttribute(s, new.
```

Figure 6.4: Eclipse Call Hierarchy Viewer showing a the call hierarchy rooted at a method named getFactory.

no direct support is provided to identifying relevant information, the resulting visualization may help a programmer in answering higher-level questions by externalizing more supporting information [114].

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 (see Figure 6.4) can be used to produce information towards answering this 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 described above supports a related feature called Autobrowsing. Autobrowsing tries to model a simple directed exploration activity between two or more selected entities. It effectively does a breadth first search and adds a source code entity to the visualization that is relevant to all of the selected entities. The process can be repeated and in some simple situations such a feature may help answer these questions about understanding connections.

In summary, tool support for answering questions in this category is limited. Of the thirteen questions, we found that four had partial support, while the rest had only minimal support. We found that often to answer these questions lower-level, possibly less refined versions of these questions (i.e., ones with better tools support) must be asked. The



```
108c122
<               for value in values
---
>               variable.each_value(level) { |value|
110,116c124,128
<                   solution[variable] = value
<                   if valid?(csp,solution,variable,level)
<                       each(csp, solution, level+1, &block)
<                   end
<                   undo_propagate(csp, solution, level) if @propagate
<               end
<               solution[variable] = UNSET
---
>                   variable.value = value
>                   each(csp, level+1, &block) if valid?(variable,csp,level)
>                   undo_propagate(csp, level) if @propagate
>               }
```

Figure 6.5: Sample output from the diff command line tools.

consequences of this include noisier results and the need to mentally put together answers, though visualization tools may make integrating this information easier.

6.4 Answering Questions About Groups of Subgraphs

In this section we discuss tool support for answering questions 34 through 44. These are all high-level questions about one or more subgraphs of a system. Including, for example, questions about how two subgraphs were related, or questions about the potential impacts of changes to a subgraph. Like questions discussed in the previous category, for the participants in our studies these questions required multiple lower-level questions to provide the necessary information and even when that information was identified, answering the intended questions could still be difficult. Tables 6.7 and 6.8 list some of the techniques and tools applicable to each of these questions, as well as the level of supported provided.

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 (especially comparing behavior) is difficult, especially comparing behavior as needed for answering question 34. For

34	**How does the system behavior vary over these types or cases?**
	Minimal support: Dynamic visualization
35	**What are the differences between these files or types?**
	Partial support: Line-based comparison tools (e.g., diff [20])
36	**What is the difference between these similar parts of the code (e.g., between sets of methods)?**
	Partial support: Line-based comparison tools
37	**What is the mapping between these UI types and these model types?**
	Partial support: Conceptual module querying [2]

Table 6.7: A summary of the techniques and tools applicable to answering questions 34 to 37, along with the level of support for each: full, partial or minimal. All of these questions are over from the fourth category and are about comparing and connecting.

questions 35 and 36, diff [20] which is a command line tool designed to show, line by line, the differences between two files, provides partial support (see Figure 6.5). However in the cases we observed that the differences (from a line by line view) were sufficiently large that diff was of limited help: *"the code is different enough that I can't just do a simple diff"* [E14].

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 (the control-flow between them, for example). Baniassad and Murphy have developed a technique and a tool called conceptual module querying, that in some situations may help with identifying these connections [2]. A conceptual module is defined by a list of source code lines and the associated tool allows programmers to ask: *how do the conceptual modules relate to each other?* If two conceptual modules are defined appropriately (one for each subgraph) then in some cases a query could yield some information of help to a programmer in answering question 37. The conceptual module query tool also allows programmers to ask: *how are the conceptual modules related to the other source code?* This query may help answer questions

38	**Where should this branch be inserted or how should this case be handled?**
	Minimal support: Visualization and browsing tools
39	**Where in the UI should this functionality be added?**
	Minimal support: Visualization and browsing tools
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 this object has been created and initialized correctly?**
	Minimal support: Visualization and browsing tools
42	**What will be (or has been) the direct impact of this change?**
	Partial support: Testing and impact analysis techniques (e.g., Chianti [74])
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

Table 6.8: A summary of the techniques and tools applicable to answering questions 38 to 44, along with the level of support for each: full, partial or minimal. All of these questions are over from the fourth category and are about changes and impacts of changes.

around how a subgraph is connected to the rest of the system, for example question 40 (*To move this feature into this code what else needs to be moved?*).

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. Unit testing and other testing techniques allow programmers to verify some aspects of the impact of changes to a code base. 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]. Note however that in our studies such a test suite was the exception, rather than the rule and that 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 exist to help programmers identify the parts of a system impacted by a change. For example, Fyson and Boldyreff show how this can be done using program understanding techniques to populate a ripple propagation graph [24] and Chianti is a tool that uses a suite of unit tests to generate a list of affected unit tests given a change to a system [74]. These techniques generate candidates for the programmer to investigate and so constitute partial support for understand 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 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 quite difficult. Tool support for this activity is limited and many of the possibly relevant research tools that exist have not been empirically evaluated in a way to support strong claims about the support provided. Despite this, we briefly describe several approaches to supporting programmers in abstracting information.

Several visualization tools (such as SHriMP, already discussed) allow subtrees and relationships to be collapsed or expanded during exploration. Other tools or techniques exist which aim to support programmers in recovering a code base's design or architecture, which has been viewed as a clustering problem [59], a CSP problem [112], a graph partitioning problem [12], a visualization and composition problem [66], and a graph matching problem [79, 80]. These techniques and tools provide a kind of abstraction over the details of a code base. They 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. The concept assignment problem is that of discovering human-oriented (high-level) concepts and assigning them to their realizations within a program [4]. Tool support for this is limited though several research tools exist including DESIRE [5], HB-CAS [26] and PAT [31]. These tools use various techniques to tag contiguous regions of code with a term that captures the *meaning* of that code. Although such an approach may help with some aspects of understanding code at a higher-level, 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.

6.5 The Gap

In this chapter we have considered techniques and tools for answering each kind of question asked by our participants. The level of support available by category is summarized in Table 6.9, with more details available in the other tables in this chapter. This summary table shows that questions in the first two categories all had at least partial support, with the majority having full support. We found that in most, though not all, situations these questions could be answered relatively directly using today's tools. The situation was quite different for questions in categories three and four. None of these questions had full support

Question Category	Full	Partial	Minimal
1. Finding an initial focus point (5 questions)	3	2	0
2. Building on a point of focus (15 questions)	12	3	0
3. Understanding a subgraph (13 questions)	0	4	9
4. Over groups of subgraphs (11 questions)	0	7	4
Total	15 (34%)	16 (36%)	13 (30%)

Table 6.9: Summary of the number of questions with *full*, *partial* or *minimal* support by question category. This information is broken out by question in Tables 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8.

with over half having only minimal support. These questions are at a relatively higher level and require programmers to consider a wider range of information and as a result these are more difficult for tools to support and more difficult for programmers to answer.

Regarding questions in category one and category two, it is important to note that often these are asked in support of higher-level questions and the particular supporting question asked will be heavily influenced by the tools available to the programmer. In general, as the actual question asked (using the available tools) gets conceptually further from the intended questions the number of questions that have to be asked will increase and the number of false positives and negatives will increase; a question asked by a programmer can be supported effectively by a given tool, but the programmer may not have been well supported in answering their intended question.

In addition to providing information about which questions (or categories of questions) can or can not be answered directly with today's programming tools, our results also suggest more general limitations with current industry and research tools. Based on our results, we believe programmers need better or more comprehensive support in three related areas: (1) support for asking higher-level questions and more precise questions, (2) support for maintaining context and putting information together, and (3) support for abstracting information and working with subgraphs at various levels of detail. These areas of support are discussed in more detail below.

1. Support for asking higher-level questions. Many programming tools support only relatively low-level questions. For example, questions that are limited to individual entities and just one type of relationship. On the other hand many of the questions asked go beyond what can be directly asked under these limitations. For example, questions about subgraphs or groups of subgraphs such as 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 which they believe will produce information that will contribute to answering their questions.

Programmers are often limited in how precise or refined their questions can be. A programmer's questions often have an explicit or implicit context or scope. In question 31 (*Which execution path is being taken* ***in this case****?*) this context is explicit. Question 33 (*What parts of this data structure are accessed in this code?*) asks about changes to a data structure but in the context of a certain section of code. Similarly, in session 1.11 participant N2 wanted to learn about the properties of an object in the context of a particular failing case of a large loop. Tools generally provide little or no support for a programmer to specify such context as a way to scope his or her questions, and so programmers ask questions more globally than they intend. Due to lack of precision or support for suitably refining queries, result sets or other information displayed by the various tools include many items irrelevant to the intended questions (i.e., false positives relative to the information the participant was seeking) and determining relevance requires additional exploration.

2. Support for maintaining context and putting information together. Most tools treat questions as if they were asked in a isolation, though we have shown that often a particular question is part of a larger process. For example, answering a higher-level question may involve multiple lower-level questions each possibly asked using different tools. Similarly, 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. Although earlier in this chapter we discussed several notable exceptions (conceptual module querying, for example), generally tools are designed to answer a specific kind of question targeting a particular programming language or type of artifact. In cases like these the burden is on the programmer to maintain the context and assemble the information which can be difficult; *"it gets very hard to think in your head how that works"* [E14]. Missing is support for bringing information together as well as support for building toward an answer. We believe also that there are missed opportunities for tools to make use of the larger context to help programmers determine what is relevant to their higher-level questions.

3. Support for abstracting information and working with subgraphs. Questions in categories three and four (that is questions about understanding subgraphs, and questions over one or more subgraphs) are distinguished from those in the other two categories by the volume and variety of information that must be considered to answer those questions. Answering these questions involves treating a number of entities as a conceptual whole and dealing with information at a number of levels. These distinguishing factors are also the factors that make answering these questions difficult as working with subgraphs at this level is not well supported by tools. Making comparisons between a number of entities is one example of an operation between subgraphs that is not well supported. As mentioned earlier, E2 spent nearly 30 minutes of session 2.2 merging the changes from one version of a pair of files into a different version of those files, because the changes were sufficiently large that diff (and therefore merge) was of limited help. In some cases, even harder is a comparison between a system's behavior in two different cases as in question 34 (*How does the system behavior vary over these types or cases?*). Further, to answer questions such as question 38 (*Where should this branch be inserted or how should this case be handled?*) requires both the right details and an overview of the right aspects of the structure; this requires support for abstracting information in a way that goes beyond collapsing subsystem nodes and composite arcs in a nested graph. Tool support for understanding systems at this level is quite limited.

6.6 Summary

Based on our results and results from previous research papers, we have investigated support provided by today's research tools for answering the 44 kinds of questions asked by our participants. We found that overall fifteen of those questions had full support, sixteen had partial support and thirteen had minimal or no support. Generally, answering questions from the first two categories was well supported, while answering questions from the third and fourth categories was much less well supported. We also observed that questions in the first two categories are at times asked precisely because there is tool support for answering them. In the process of this investigation, we have identified several areas in which programmers need better support, including support for asking higher-level and more precise questions, support for maintaining context and putting information together, and finally support for abstracting information and working with subgraphs at various levels of detail. Our results suggest that improvements in these areas will move programming tools closer to programmers' questions and the process of answering those questions.

Chapter 7

Discussion

The previous two chapters have presented the results of our analysis of the data collected from our two studies. This presentation has included a discussion of areas where we have found tool support to be missing for activities around answering questions (see Section 6.5). The goal of this chapter is to provide a brief discussion of these studies and results. First, we discuss the implications of our results in Section 7.1. This discussion includes suggestions for programming tool features to better support the activities we observed around answering those questions. To make these feature suggestions concrete we describe features for a hypothetical search tool. Second, we discuss our findings in relation to some earlier work (see Section 7.2). Third, we discuss the limitations of our studies which affect how our results should be interpreted (see Section 7.3). Finally, we discuss possible future studies that could be used to build on our research (see Section 7.4).

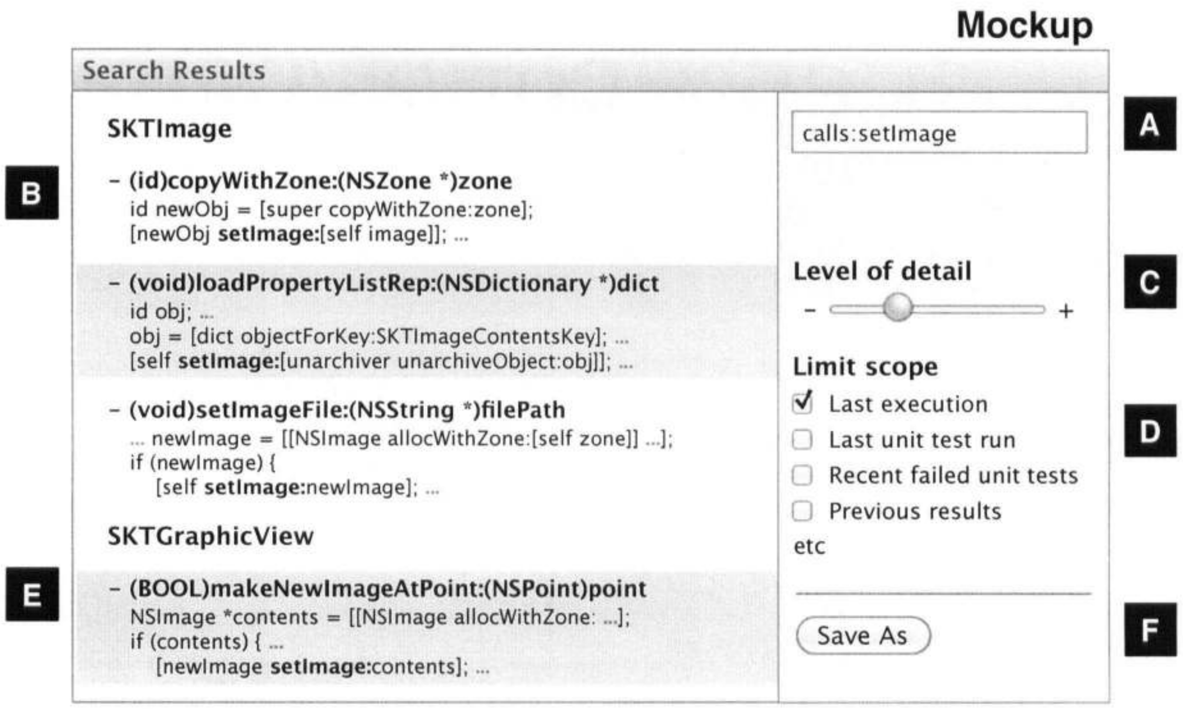
7.1 Implications

Our analysis has identified particular questions that can not easily be answered using current tools. We observed that the most difficult questions to answer are often the higher-level questions–those about one or more subgraphs of a system. One very effective way to get an answer to such a high-level question is to "ask somebody who knows" [48]. People have an ability to make a sketch or provide an explanation that captures just the right details; abstracting and crossing boundaries. Their tools are natural language, metaphors and pictures. Using these people can relatively effectively answer the types of questions that are the most difficult to answer using current tools. Though there are limitations with people's abilities here: *"there are lots of different metaphors [used to describe this code base]* *but it ends up diving in to details so quickly that the metaphors are often lost"* [E2]. The availability of documentation in our two studies was limited. Based on our results, we believe that improving the ability of programmers to document and communicate about parts of their code would be valuable. This support should focus on subgraphs of the system including issues around "why" and "how best" kinds of questions.

When neither documentation nor an expert is available to help answer a programmer's questions, he or she heavily relies on tools to help with activities around answering those questions. Our results point to several aspects of the gap between the support programmers need in answering their questions and the support that tools provide (see Section 6.5). Here we explore possible ways that tools can be designed to better support the process of answering questions.

Based on support that we have shown to be missing from current tools, we believe that programmers need tools that support questions or views that cross relationship types and artifact types. Tools need to support more options for scoping queries (possibly integrating techniques such as slicing or feature location techniques) to increase the relevance of information presented. Tools that present lists or trees of source code entities could show additional information (some source code details, for example) to make it easier for programmers to determine what is relevant and to increase the information content of the view. Search results or other views should maintain context between searches or other actions. Examples of contexts that could be profitably used include: visited entities, previous search results, modified entities and executions of the program. These kinds of contexts could be made explicit and used between tools. If made explicit these represent groups of entities that tools could support operations over (intersecting, union, comparison and connecting, for example) and generally should support working with and seeing together. Finally, in bringing together more information, tools should support flexibly combining appropriate details and overviews of that information.

To make these suggestions more concrete, next we describe features of a hypothetical search tool. The tool is mocked up in Figure 7.1. On the left are the search results. On the



SKTImage

```
- (id)copyWithZone:(NSZone *)zone
    id newObj = [super copyWithZone:zone];
    [newObj setImage:[self image]]; ...

- (void)loadPropertyListRep:(NSDictionary *)dict
    id obj; ...
    obj = [dict objectForKey:SKTImageContentsKey]; ...
    [self setImage:[unarchiver unarchiveObject:obj]]; ...

- (void)setImageFile:(NSString *)filePath
    ... newImage = [[NSImage allocWithZone:[self zone]] ...];
    if (newImage) {
        [self setImage:newImage]; ...
```

SKTGraphicView

```
- (BOOL)makeNewImageAtPoint:(NSPoint)point
    NSImage *contents = [[NSImage allocWithZone: ...];
    if (contents) { ...
        [newImage setImage:contents]; ...
```

calls:setImage

Level of detail: − ○ +

Limit scope
- [✓] Last execution
- [ ] Last unit test run
- [ ] Recent failed unit tests
- [ ] Previous results
- etc

Figure 7.1: A mockup of a search tool illustrating some of the suggestions in our results. On the left are the search results (from an Objective-C code base) and on the right are controls that affect the set of results shown. Parts of the mockup labeled A through F are described in Section 7.1.

right are controls that affect the set of results shown. Parts of the mockup are labeled A through F and are described below.

A. The search box with support for simple keyword searches as well as support for more complicated searches including searches over multiple relationship types. The very simple example in the diagram represents a search for all methods that call a method named setImage.

B. A summary of each item (method in this case) returned from the search organized by type. A slice algorithm based on the primary reason for the items inclusion in the result set could be used to scale this summary in an intelligent way. Since this search was for all callers to setImage, the summary (i.e., the slice) of each method would center on the line of code containing that call.

C. A slider allowing the programmer to determine the size of the summary (i.e., the slice) to show. The size could range from just the method name to the entire method body. This extra information can be used to determine what is relevant and generally to understand aspects of the system relative to the current search.

D. This search tool could support various ways to limit the scope of searches. In the case shown in the diagram, only methods that both call setImage and were encountered in the most recent execution of the system would be included in the result set. Other scopes shown in the figure are based on unit test runs and previous searches. Other possibilities could include scopes based on output from feature location tools (to scope searches by feature), navigation or modification history, and revision control history.

E. Information about context can be provided by highlighting entities in the result set. For example, different color highlighting could emphasize entities returned in previous searches, modified entities or previously visited entities. Two methods (loadPropertyListRep and makeNewImageAtPoint) are highlighted in this example.

F. Results sets can be manually modified (items removed from the set, for example) and the resulting set can be given a name and saved. This save feature could allow a programmer to specify a group of relevant entities. In this way an arbitrary group of entities can be revisited, documented and shared, used as a scope for future searches, etc.

7.2 Related Work

In Section 2.1.3, we describe earlier work involving studies of programmers and the questions they ask. In this section we compare our findings with those of the most related of this earlier work.

Erdos and Sneed suggest, based on their personal experience, that seven questions need to be answered for a programmer to maintain a program [17]. The first six questions identified by Erdos and Sneed map in a very direct way with questions asked by our participants. For example, their first question (Where is a particular subroutine/procedure invoked?) is captured by our question 12 (*Where is this method called or type referenced?*). However, their seventh question (What are the inputs and outputs of a module?) is not captured by a question in our catalog since we observed no evidence of this question (or similar questions) being asked by our participants. This may be due to the fact that we observed programmers working on specific change tasks, rather than on more general understanding tasks.

Letovksy's work on *inquiry episodes* [56, 57] focuses on "conjectures" which are questions along with a hypothesis of the answer. In our studies we also observed some evidence of this form of questioning, though most of the questions we observed were not formulated in this way and so we have not made this distinction. Letovsky does not extensively catalog the conjectures asked by his participants, instead he discusses five categories of conjectures (*why*, *how*, *what*, *whether* and *discrepancy*). This represents an alternative categorization to the one we have proposed in Section 5.1. The differences between our categorizations stem from differences in our goals: Letovsky's work aims for a description of the mental processes involved, while our work aims for a characterization of the information needs associated with the questions as we believe such a characterization will be valuable for tool builders.

As described in Section 2.1.2, our work also relates to an earlier effort by Von Mayrhauser and Vans to inform the design of tools based on their program comprehension work [102]. This earlier work identifies several information needs based on the tasks and subtasks that make up their model of program comprehension. In contrast, our work has characterized the information needed by programmers based on the questions our participants asked and the portion of the system relevant to answering those questions. As a result of the differences between our approaches, the similarities between the information needs proposed by Von Mayrhauser and Vans are generally quite small. Also, our findings around programmers' processes are at quite different levels. Their analysis has aimed to produce a description of the mental processes of programmers, while we have aimed to produce a description of the work processes involved, including their use of tools. Future research should aim to understand the connections between the mental processes and the work processes.

7.3 Limitations

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 here.

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 [101], 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 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 non-local 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 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 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 which covered a wide range of tools.

Our analysis of the level of tool support for answering questions (presented in Chapter 6) is limited in two important ways. First, it is possible we have overlooked a particular research tool applicable to answering one of the questions our participants asked, and covering every tool has not been our goal. This limitation means that it is possible that one or more questions have a higher level of support than noted in our analysis. Second, as many of the tools 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. For this reason there is a certain degree of subjectivity between what constitutes full support, partial support and minimal support, and for this reason one or more of the questions arguably have a higher or lower level of support than noted in our analysis. However we believe that these limitations had at most a small effect on particular details of our analysis and have not impacted the overall findings.

7.4 Follow-up Studies

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 7.2 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 first

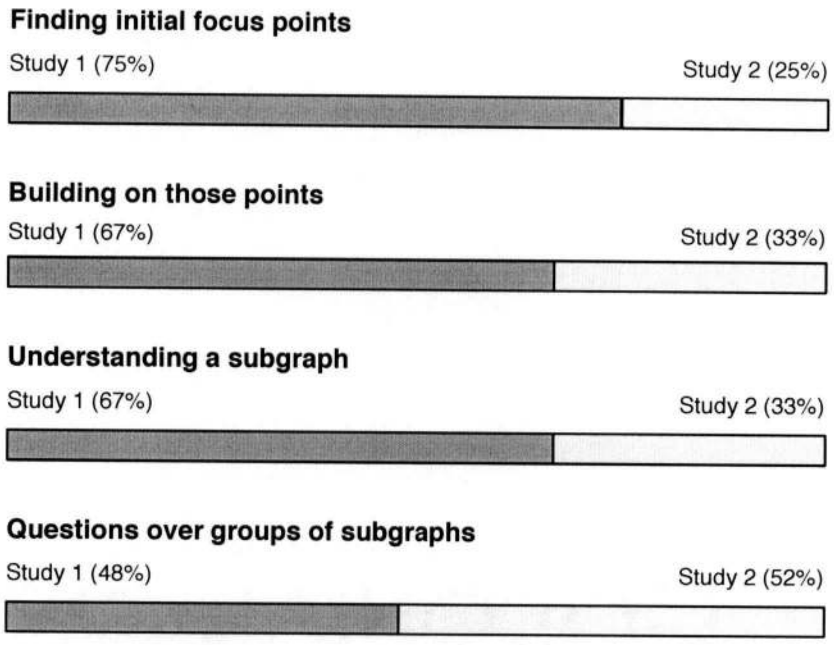


Figure 7.2: The distribution of observed question occurrences across the two studies by category. An occurrence consists of one or more questions from a category being asked in a session. For example, 25% of the sessions that featured a question from the first category were from study two.

study were newcomers to the code they were working on while participants in the second study were working with code they had experience with. However, other factors may have also contributed. In this section 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.

The two studies we have performed and analyzed have provided a wealth of information around programmers asking and answering questions. At the same time our analysis 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.

In addition to studies that build on our work we believe that our results provide support for user studies that aim to evaluate tools. Designing a study that has both reasonable overhead and can be shown to capture a realistic and important situation can be challenging for researchers. We believe that information such as questions that programmers need to answer as well as the activities involved in answering those questions can be valuable for developing lighter weight studies that can be shown to be applicable to programming activities in the real world.

Chapter 8

Summary

A significant amount of research in software engineering has been based on researchers' own intuitions and experiences about problems in software development, rather than on empirical results. Much of the empirical research that has been conducted has focused on smaller programs using older methods and tools. The result of this has been that programming tools have not helped as much as they might and many aspects of activities around programming are not as well understood as they need to be. For example, 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? The goal of this research has been to provide an empirical foundation for tool design based on an exploration of what programmers need to understand and of how they use tools to discover that information while performing a change task.

To this end, 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. The participants in the first study were observed as they worked on assigned change tasks to a code base that was new to them. This code base comprises roughly 60K lines of Java code. The participants in the second study were all professional programmers working for the same company. These participants were observed as they worked on their own change task to a software system for which they had responsibility. These system varied from relatively modest sized ones (about 20K lines of code) to very large ones (over one million lines of code). Participants in the first study used the Eclipse Development Environment while participant in the second study used the tools that they normally used, which varied significantly (Emacs, Visual Studio, etc). 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.

To structure our data collection and the analysis of that data, we have used a grounded theory approach. This collection and analysis was an iterative process of discovering questions in the data and exploring similarities, connections and differences among those questions. In the process we developed generic versions of the questions, which slightly abstract from the specifics of a particular situation and code base. We also analyzed the behavior we observed around answering those questions both at the particular question level and at the category level. To build on this, and to increase the generalizability of our results, we have performed additional analysis of tool support for answering each type of question. This analysis considered a wide range of tools (beyond those used by our participants) and allowed us to explore the level of support today's tools provide for answering the questions our participants asked.

This research has made four key contributions. The first contribution is an empirically based catalog of the 44 types of questions asked by the participants of the two studies. These are slight abstractions over the many specific questions and situations we have observed. These abstractions have allowed us to compare and contrast the questions asked as well as to gather some simple frequency data for those questions. The questions asked range from simple low-level questions such as *Who implements this interface or these abstract methods?* to much more involved questions such as Why isn't control reaching this point in the code?. Although not a complete list of questions, it illustrates the sheer variety of questions asked and provides a basis for an analysis of tool support for answering questions. To our knowledge this is the most comprehensive such list published to date.

The second contribution is a categorization of those 44 types of 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 which aim at understanding subgraphs of a system, and the final category groups questions over one or more such subgraphs. Although other categorizations of the questions are possible, we have selected this categorization because these categories show the types and scope of information needed to answer the questions, capture some intuitive sense of the various levels of questions asked, and make clear various kinds of relationships between questions, as well as certain observed challenges around answering those questions.

Third, an analysis of the observed process of asking and answering questions, which provides valuable context for the questions we report. We have shown that many of the questions asked were closely related. For example, some lower-level questions were asked as part of answering higher-level questions. We have shown that the questions a participant asked often mapped imperfectly to questions that could be answered directly using the available tools. For example, at times the questions programmers ask using current tools are more general than their intended questions. We have also shown that programmers, at times, need to mentally combine result sets or other information from multiple tools to answer their questions. These results contribute to our understanding both about how programmers answer their questions and the challenges they face in doing so.

The final contribution from our research is an analysis of the support existing tools (both industry tools and research tools) provide for answering each kind of question. We consider which questions are well supported and which are less well supported. We also generalize this information to demonstrate a gap between the support tools provide and that which programmers need. In particular we show that programmers need improved support for asking higher-level questions and more precise questions, support for for maintaining context and putting information together, and support for abstracting information and working with subgraphs of source code entities and relationships. We hope these results will provide motivation and a foundation for the design of future programming tools.

These results provide a more complete understanding of the information needed by programmers performing nontrivial change tasks, and of how programmers use tools to discover that information. These results have several implications for tool design. Our results point to the need to move tools closer to programmers' questions and the process of answering those questions and also suggest ways that tools can do this, for example, by maintaining and using more types of context and by providing support for working with larger and more diverse groups of entities and relationships. More generally, 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.

Bibliography

[1] Brian De Alwis and Gail C. Murphy. Using visual momentum to explain disorientation in the eclipse IDE. In *Proceedings of the IEEE Symposium Visual Languages and Human Centric Computing (to appear)*, 2006.

[2] Elisa Baniassad and Gail Murphy. Conceptual module querying for software engineering. In *Proceedings of the International Conference on Software Engineering*, pages 64–73. IEEE, 1998.

[3] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE13(12):1278–1296, 1987.

[4] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. The concept assignment problem in program understanding. In *Proceedings of the International Conference on Software Engineering*, 1993.

[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[6] Andrew P. Black and Nathanael Scharli. Traits: Tools and methodology. In *Proceedings of the International Conference on Software Engineering*, pages 676–686, 2004.

[7] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer Magazine*, 1987.

[8] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9:737–751, 1977.

[9] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[10] Brown. *Algorithm Animation.* MIT Press, 1988.

[11] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[12] R. Chanchlani. Software architecture recovery and design using partitioning. Master's thesis, University of Waterloo, 1996.

[13] Kunrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 241–249. IEEE, 2000.

[14] Jacob L. Cybulski and Karl Reed. A hypertext based software-engineering environment. *IEEE Software*, 9(2):62–68, 1992.

[15] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of ACM 2005 Symposium on Software Visualization*, pages 183–192. ACM, 2005.

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

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

[18] Nick V. Flor and Edwin 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. Ablex Publishers, 1991.

[19] Clif Flynt. *Tcl/Tk: A Developer's Guide.* Morgan Kaufmann, 2nd edition, 2003.

[20] Free Software Foundation, http://www.gnu.org/software/diffutils/manual/diff.html. *Comparing and Merging Files*, May 2002.

[21] Free Software Foundation, http://www.gnu.org/software/grep/doc/grep.html. *Grep, print lines matching a pattern*, January 2002.

[22] Free Software Foundation, http://sources.redhat.com/gdb/current/onlinedocs/gdb.html. *Debugging with GDB*, September 2006.

[23] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tools for distributed software development teams. In *Proceedings of the International Conference on Software Engineering*, pages 387–396, 2004.

[24] M. Joanna Fyson and Cornelia Boldyreff. Using application understanding to support impact analysis. *Journal of Software Maintenance Research and Practice*, 10(2):93–110, December 1998.

[25] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine Publishing, 1967.

[26] Nicolas Gold. Hypothesis-based concept assignment to support software maintenance. In *IEEE International Conference on Software Maintenance*, pages 545–548. IEEE, 2001.

[27] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishing Company, 1984.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley Longman, Inc., 2000.

[29] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[30] Graeme S. Halford, Rosemary Baker, Julie E. McCredden, and John D. Bain. How many variables can humans process? *Psychological Science*, 16(1):70–76, January 2005.

[31] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, January 1990.

[32] Sivakumar Harinath and Stephen R. Quinn. *Professional SQL Server Analysis Services 2005 with MDX*. Wrox, 1st edition, 2006.

[33] Mark Harman, Nicolas Gold, Rob Hierons, and Dave Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering*, pages 11–21. IEEE, 2002.

[34] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, 3rd edition, 2004.

[35] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison Wesley Professional, 2nd edition, 2006.

[36] Scott Henninger. Retrieving software objects in an example-based programming environment. In *Proceedings of the 14th International ACM SIGIR Conference on Automated Software Engineering*, pages 408–418, 1991.

[37] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The university of washington illustrating compiler. In *Proceedings of the ACM SIGNPLAN'90 Conference on Programming Language Design and Implmentation*, pages 223–233, 1990.

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

[39] Darren Hiebert. *Ctags*. http://ctags.sourceforge.net/ctags.html, March 2004.

[40] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[41] Carolyn J. Hursch. *SQL: The Structured Query Language.* Windcrest, 2nd edition, 1991.

[42] Suhaimi Ibrahim, Horbik Bashah Idris, and Aziz Deraman. Case study: Reconnaissance techniques to support feature location using recon2. In *Proceedings of the Software Engineering Conference*, pages 371–378, 2003.

[43] Object Technology International, Inc. Eclipse platform technical overview. White Paper, 2001.

[44] Daniel Jackson. Aspect: An economical bug detector. In *Proceedings of the International Conference on Software Engineering*, pages 13–22, 1994.

[45] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings SIGSOFT Foundations of Software Engineering Conference (FSE)*, pages 2–10, 1994.

[46] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *proceedings of the international conference on aspect-oriented software development*, pages 178–187, 2003.

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

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

[49] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[50] Martin Klaus. Simplifying code comprehension for legacy code reuse. *Embedded Developers Journal*, April 2002.

[51] Andrew J. Ko, Htet Htet Aung, and Brad 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. ACM Press, 2005.

[52] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program failures. In *CHI*, pages 151–158, 2004.

[53] Jurgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI conference on Human factors in computer systems: Reaching through technoloty*, pages 125–130. ACM Press, 1991.

[54] Wojtek Kozaczynski, Stanley Letovsky, and Jim Ning. A knowledge-based approach to software system understanding. In *Knowledge-Based Software Engineering Conference*, 1991.

[55] Eiji Kuwana and James D. Herbsleb. Representing knowledge in requirements engineering: An empirical study of what software engineers need to know. In *IEEE International Symposium on Requirements Engineering*, 1993.

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

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

[58] David Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Proceedings of the Conference on Empirical Studies of Programmers*, pages 80–98. Ablex Publishers, 1986.

[59] Spiros Mancoridis, Brian Mitchell, C. Rorres, Yih-Farn Chen, and Emden Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–53. IEEE Computer Society, 1998.

[60] Robert Mays. Power programming with the lemma code viewer. Technical report, IBM TRP Networking Labratory, 1996.

[61] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):183–210, 1956.

[62] Tim Miller and Paul Strooper. A framework and tool support for the systematic testing of model-based specifications. *ACM Transactions on Software Engineering and Methodology*, pages 409–439, 2003.

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

[64] Sougata Mukherjea and John T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM TransactIons on Computer-Human Interaction*, 1(3):215–244, 1994.

[65] Hausi A. Muller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE Computer Society Press, April 1988.

[66] Hausi A. Muller, Mehmet A. Orgun, Scott R. Tilley, and James S Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993. Uses Rigi.

[67] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[68] Chuck Musciano and Bill Kennedy. *HTML & XHTML: The Definitive Guide.* O'Reilly Media, 6th edition, 2006.

[69] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualzation for novice pascal programmers. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 192–198, 1988.

[70] Patrick D. OBrien, Daniel C. Halbert, and Michael F. Kilian. The trellis programming environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, page 91102, 1987.

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

[72] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[73] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, 1990.

[74] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Object-Oriented Systems, Languages, Programming, and Applications (OOPSLA)*, pages 432–448. ACM Press, 2004.

[75] Tamar Richner and Stephane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the International Conference on Software Maintenance*, pages 13–22, 1999.

[76] Scott P. Robertson, Erle F. Davis, Kyoko Okabe, and Douglas Fitz-Randolf. Program comprehension beyond the line. In *Proceedings of the IFIP TC13 third international conference on human-computer interaction*, pages 959–963. ACM, 1990.

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

[78] Martin P. Robillard and Gail C. Murphy. Feat. a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 822–823, 2003.

[79] Kamran Sartipi and Kostas Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 408–419. IEEE, 2001.

[80] Kamran Sartipi and Kostas Kontogiannis. Pattern-based software architecture recovery. In *Proceedings of the Second ASERC Workshop on Software Architecture*, 2003.

[81] Namir Clement Shammas. *Windows Batch File programming.* Computing Mcgraw-Hill, 2nd edition, 1995.

[82] Ben Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[83] Benjamin Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop Publishers Inc., 1980.

[84] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the SIGSOFT Foundations of Software Engineering Conference (to appear)*, 2006.

[85] Jonathan Sillito, Kris De Volder, Brian Fisher, and Gail Murphy. Managing software change tasks: An exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering*, 2005.

[86] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of CASCON'97*, pages 209–223, 1997.

[87] Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *OOPSLA '05 Eclipse Technology eXchange (ETX) Workshop*, 2005.

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

[89] Elliot Soloway, Robin Lampert, Stanley Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.

[90] Joe Steffen. Interactive examination of a c program with cscope. In *Proceedings of the USENIX Winter Conference*, pages 170–175, 1985.

[91] Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools.* PhD thesis, Simon Fraser University, 1998.

[92] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Muller. Cognitive design elements to support the construction of a mental model during software visualization. In *International Workshop on Program Comprehension*, pages 17–28, 1997.

[93] Margaret-Anne D. Storey, F. David Fracchia, and Hausi 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.

[94] Margaret-Anne D. Storey, Hausi A. Muller, and Kenny Wong. Manipulating and documenting software structures. *Software Visualization*, 1996.

[95] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[96] Anselm L. Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for developing Grounded Theory.* Sage Publications, 1998.

[97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman, Inc., 2nd edition, 1991.

[98] Warren Teitelman and Larry Masinter. The interlisp programming environment. *IEEE Computer*, 14(4):25–34, 1981.

[99] Manfred Thuring, Jorg Hannemann, and Jorg M. Haake. Hypermedia and cognition: Designing for comprehension. *Communications of the ACM*, 38(8):57–66, 1995.

[100] Scott R. Tilley, Santanu Paul, and Dennis B.Smith. Towards a framework for program understanding. In *Proceedings of the Workshop on Program Comprehension*, pages 19–28, 1996.

[101] Maarten W. van Someren, Yvonne F. Barnard, and Jacobijn A.C Sandberg. *The Think Aloud Method; A Practical Guide to Modelling Cognitive Processes.* Academic Press, 1994.

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

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

[104] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework.* PhD thesis, Simon Fraser University, 2002.

[105] Andrew Walenstein. Theory-based cognitive support analysis of software comprehension tools. In *Proceedings of the International Workshop on Program Comprehension*, pages 75–84. IEEE Computer Society, 2002.

[106] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE, 1981.

[107] A. Keyton Weissinger. *ASP in a Nutshell.* O'Reilly Media, 2nd edition, 2000.

[108] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.

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

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

[111] David D. Woods. Visual momentum: A concept to improve the cognitive coupling of person and computer. *International Journal of Man-Machine Studies*, 21:229–244, 1984.

[112] Steven G. Woods, Alexander E. Quilici, and Qiang Yang. *Constraint-Based Design Recovery for Software Engineering: Theory and Experiments*. Springer, 1997.

[113] Jiajie Zhang. The nature of external representations in problem solving. *Cognitive Science*, 21(2):179–217, 1997.

[114] Jiajie Zhang and Donald A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.

Appendix A

Ethics Board Certificate of Approval

The studies for this research were approved by the University of British Columbia's Behavioral Research Ethics Board. For a copy of the original *Certificate of Approval* please see the next page.