Developers Ask Reachability Questions

Thomas D. LaToza
Institute for Software Research
School of Computer Science
Carnegie Mellon University
tlatoza@cs.cmu.edu

Brad A. Myers
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
bam@cs.cmu.edu

ABSTRACT

A *reachability question* is a search across feasible paths through a program for target statements matching search criteria. In three separate studies, we found that reachability questions are common and often time consuming to answer. In the first study, we observed 13 developers in the lab and found that half of the bugs developers inserted were associated with reachability questions. In the second study, 460 professional software developers reported asking questions that may be answered using reachability questions more than 9 times a day, and 82% rated one or more as at least somewhat hard to answer. In the third study, we observed 17 developers in the field and found that 9 of the 10 longest activities were associated with reachability questions. These findings suggest that answering reachability questions is an important source of difficulty understanding large, complex codebases.

Categories and Subject Descriptors

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

General Terms

Human factors.

Keywords

Program comprehension, empirical study, software maintenance, code navigation, developer questions

1. INTRODUCTION

A central goal of software engineering is to improve developers' productivity and the quality of their software. This requires an efficient and effective way to explore code since most developers will encounter code with which they are not familiar. Understanding code in modern codebases is challenging because of the size and complexity of the codebase, and the use of indirection. For example, many modern codebases use callbacks and events to connect modules or communicate with external frameworks. While use of indirection enables reuse, it also makes understanding relationships between behaviors more challenging. For example, an analysis of code in Adobe's desktop applications found that one third of the codebase is devoted to event handling logic which in turn caused half of the reported bugs [15]. Successfully coordinating dependencies among effects in loosely connected modules can be very challenging [5].

To better understand how developers understand large, complex codebases, we conducted three studies of developers' questions during coding tasks. Surprisingly, we discovered that a significant portion of developer's work involves answering what we call *reachability questions*. A reachability question is a search across all feasible paths through a program for statements matching search criteria. Reachability questions capture much of how we observed developers reasoning about causality among behaviors in a program.

Consider an example from our first study: after proposing a change, a developer sought to determine if it would work before committing to implementing it. To do so, he wanted to determine "all of the events that cause this guy to get updated". While he was aware that a call graph exploration tool could traverse chains of method calls, this did not directly help. Upstream from the update method was a bus onto which dozens of methods posted events, but only a few of these events triggered the update. Existing call graph tools are unable to identify only those upstream methods sending the events triggering the update of interest. Unable to answer the question in any practical way, he instead optimistically hoped his guess would work, spent time determining how to reuse functionality to implement the change, edited the code, and tested his changes before learning the change would never work and all his effort had been wasted.

We found that many of the problems that developers experience understanding code arise from difficulties answering reachability questions. In a lab study of modifications to complex, unfamiliar code, developers often inserted defects because they either could not successfully answer reachability questions or made false assumptions about reachability relationships. A survey of developers that asked about 12 rearchability questions revealed that, on average, 4.1 of these were thought to be at least somewhat hard to answer. And these questions were not limited to inexperienced developers or those new to a codebase: neither professional development experience nor experience with their codebase made these questions less frequent or easier to answer. Reachability questions can be time consuming to answer. In a field study, developers often spent tens of minutes answering a single reachability question.

This paper presents data about reachability questions gathered from over 470 developers and over 70 hours of direct observations of coding tasks. We first review related work and formally define reachability questions. Next, we present the method and results of each of three studies in turn and discuss their findings. Finally, we

ICSE '10, May 2-8, 2010, Cape Town, South Africa
Copyright © 2010 ACM 978-1-60558-719-6/10/05 ... $10.00

discuss the implications of these findings for helping developers more effectively understand code.

2. RELATED WORK

It has long been known that control and data flow are central to how developers mentally represent programs [3]. Studies of program comprehension have found that developers begin understanding small programs by constructing a mental model of control flow [16]. A number of studies have applied the idea of information foraging to describe how developers navigate code [14]. Beginning at an origin method, developers use cues such as method names to pick which of many calls to traverse towards targets, and remember information they collect [11]. Simulations of code navigation have focused on understanding exactly how developers decide which calls to traverse. For example, one study found that words contained in a bug report were usually sufficient to explain which calls developers decide to traverse [14]. Developers must also remember what they find. Difficulties here have been found to lead to information loss, poor representation choices, and problems returning to points where information was previously found [9]. These studies illustrate the central importance of navigating and exploring code to coding tasks.

Several recent studies have observed developers at work in coding tasks to identify information needs or questions associated with development activities. One study identified 21 questions developers ask about interactions in code, artifacts, and teammates [10]. We believe that one-third of these questions are questions which developers might answer by asking reachability questions. Some of the other questions were related to communication with teammates, maintaining awareness of changes, and reasoning about design. Another study identified 44 questions specifically about code [17]. They reported that developers refine their questions from higher-level questions, such as the implications of their changes, into lower-level questions that can be more directly answered using the development environment. Many of the challenges developers experienced stemmed from problems separating task-relevant results from the many task-irrelevant results that the tools in the development environment produced. Several of the questions they identified were specifically about control or data flow. Interestingly, of the questions identified as not well supported by existing tools, 52% were questions we believe developers might answer by asking reachability questions.

Several studies have observed developers using existing tools for understanding control flow to produce recommendations for future tools that would more effectively support developers' needs. One study observed developers using a UML tool while editing code [4]. In addition to identifying several usability problems, a key recommendation was to better support selecting task-relevant items in the reverse engineered view to prevent wasted time understanding task-irrelevant items. They also saw the need for much more automated support for reverse engineering sequence diagrams. Another study failed to find much use of detailed large-scale maps of code hung on walls near developers' offices [2]. Designed to be useful for all possible tasks, these diagrams had both too much and too little information – developers required many details but only those that were task-relevant. The authors conclude that diagrams providing concise and targeted answers to situation-relevant questions were more likely to be useful than general-purpose diagrams. Another study observed several students using a UML sequence diagram tool in the lab [1]. Participants specifically requested the ability to rapidly configure the diagram to filter or search for items and to easily hide items that were determined to be uninteresting. Overall, these studies suggest that developers could benefit greatly from diagrams that are more focused on task-relevant items, but the studies provide little guidance on what developers find task-relevant.

Only a few studies have attempted to measure the time developers spend on development activities. In one study [18], 8 developers were observed for an hour each. They most frequently executed UNIX commands, followed by reading the source, loading or running software, and reading or editing notes. In a later study, developers at Microsoft, when surveyed about their use of time, reported spending nearly equal amounts of time communicating, understanding, writing new code, editing old code, and on non-code related activities [13]. Developers reported spending somewhat less time designing, testing, and on other activities. A detailed study of 10 students at work on a lab task found 22% of time spent reading code, 20% editing code, 16% navigating dependencies, 13% searching, and 13% testing [9]. Thus, developers are spending significant time trying to understand code.

3. DEFINITIONS

From preliminary analysis of related work and results from our studies, it seemed clear that developers ask a class of questions that had not previously been explicitly characterized – *reachability questions*. But exactly which questions do these include? While we had many examples, often in developers' own words, a formalism would unambiguously show how each was a reachability question and highlight relationships between similar questions. So we used our examples to design a formalism for reachability questions which we describe here. Although we developed it chronologically after the studies, we present it first in this paper to use it to describe the questions we observed (see Tables 1 and 2).[1]

3.1 Reachability questions

Intuitively, a reachability question is a search across feasible paths through a program for target statements matching search criteria. Thus, a reachability question consists of two parts: the paths to search and the search criteria specifying the statements to find.

Reachability questions represent feasible paths as a set of concrete traces *TR*. A concrete trace *tr* is a list of <*s, env*> tuples, where *s* is a statement and *env* maps every variable in *s* to a value. *traces(p, O, D, C)* is the set of all concrete traces in a program *p* from an origin statement *o* in the set *O* to a destination statement *d* in the set *D* which satisfy all the filtering constraints *c* in *C*. *O, D,* and *C* can be left unspecified by using a *?* (although at least one of *O* or a *D* must be specified). Questions without an origin are called *upstream* reachability questions while questions with an origin (and optionally a destination) are *downstream* reachability questions. *C* is a set of filtering constraints *c*, where *c* is a tuple <*s, x, const*> specifying a value for a variable *x* in *s*. *x* and *const* can be left unspecified (?) to find only the traces containing *s*.

There are two types of reachability questions: *find* and *compare*. *find SC in TR* finds the portion of each *tr* in the set of traces *TR* that match search criteria *SC*. A search criteria function, given attributes describing a set of statements, generates a set of statements *SC*. Table 1 lists search criteria functions we observed in our studies. A reachability question then matches *SC* against each <*s, env*> tuple in a trace *tr* to generate new traces containing only

[1] The formalism was designed with the assistance of Jonathan Aldrich.

Function	Finds the set of statements that:
grep(str)	include text matching the string *str*
reads(F), *writes(F)*	read / write a field *f* in the set of fields *F*. *FIELDS* is the set of all fields in the program.
stmts(T)	are in a type *t* in the set of types *T*
stmts(M)	are in a method *m* in the set of methods *M*
callers(M)	are callsites of a method *m* in the set of methods *M*
callees(M)	are method declaration statements of methods invoked by a method *m* in the set of methods *M*
ends	are method calls to framework methods without source or method declaration statements with no callers which may be callbacks
dDepend(s, x)	*x* in *s* has a data dependency on. *dDepend(s, x)** finds the transitive closure including transitive data dependencies.

Table 1. Search criteria functions describing statements for which developers searched (see tables 2 and 3).

tuples where *s* is in *SC*.

compare(TR_a, TR_b) : TR_{common}, TR_1, TR_2 compares sets of traces. Compare first, by an unspecified method, attempts to match each tr_a in TR_a to a corresponding trace tr_b in TR_b. When such a match is found, compare then attempts to match tuples $<s_a, env_a>$ in tr_a to corresponding tuples $<s_b, env_b>$ in tr_b. This generates three new lists: tr_{common} which contains an ordered list of tuples that matched, and tr_1 and tr_2 which contain an ordered list of tuples in tr_a and tr_b that did not match. TR_1 and TR_2 also contain traces in TR_a and TR_b for which no match could be found.

3.2 Comparison to slicing

Many tools have been designed to help developers explore programs by finding sets of statements. One technique used by many tools is *slicing* [7][19][21][22]. Slicers find statements connected by either data dependencies or control dependencies. Data dependency *dDepend*(s_1, *x*) finds the set of statements *S* where each s_2 in *S* may have last defined a variable *x* used in s_1. A *control dependency* exists from s_1 to s_2 if s_2 controls if s_1 does or does not execute. *cDepend*(s_1) finds all such control dependencies of s_1. A *(backward) static slice* [22] is simply the transitive closure of the union of these two relations: $(dDepend(s_1, x) \cup cDepend(s_1))^*$. In a highly influential study, Weiser [21] found that developers debugging better remembered a static slice related to the bug than either an unrelated slice or an arbitrary portion of the program. This suggested that developers follow slices when using the strategy of debugging backwards from an error to a bug.

Building on this work, many variations on slicing have been proposed [7][20]. A *forward slice* finds control and data dependencies forwards rather than backwards: $(fdDepend(s_1, x) \cup fcDepend(s))^*$. A *dynamic slice* finds control and data dependencies in a particular execution. Like reachability questions with a filtering constraint, *conditioned static slices* find dependencies across paths which satisfy a constraint. A *thin slice* finds data dependencies $dDepend(s_1, x)^*$ while excluding data dependencies at pointer dereferences [19]. A *chop* intersects statements in a forwards slice on *x* at s_1 with a backwards slice on *y* at s_2: $(fdDepend(s_1, x) \cup fcDepend(s_1))^* \cap (dDepend(s_2, y) \cup cDepend(s_2))$. The central idea of all slicing techniques applied to code exploration is to use control and data dependencies to find statements answering a developer's question.

Reachability questions differ from slicing in many ways. First, by searching over the set of all concrete traces, reachability questions exclude infeasible paths. Static slicing techniques are typically defined as a *may* analysis where statements may be dependent only through infeasible paths that never execute. However, much of the work done on improved slicing has focused on eliminating infeasible paths by, for example, introducing context sensitivity [20]. Thus, the difference is only that a reachability question specifies a fully precise answer whereas slices specify answers of any precision. A second difference is that reachability questions find portions of traces where statements can occur multiple times, while static slicers often find a subset of the program where statements occur exactly once. Dynamic slicers search over traces, but only a single trace rather than the set of all traces.

An important difference between reachability questions and slicing is that a reachability question is a search across control flow paths rather than dependencies. By design, the set of statements in a slice will always be a subset of the statements across control flow paths: statements that are not dependent are not included. Slices correspond to questions about influence: "Why did this execute?" (control dependency), or "Where did this value come from?" (data dependency). In contrast, control flow captures questions about what happens before ("What are the situations in which?") or after ("What does this do?"). When developers ask a question about control flow, the slice may not include the statements answering their question. And while our reachability question formalism includes searches for data dependencies, we observed only 1 example of such a question out of the 17 important reachability questions we found (tables 2 and 3).

The most important difference between slicing and reachability questions is that a reachability question is a search for a set of statements described by any of a wide variety of search criteria. Consider an example from study 1: a developer wondered why calling a method *m* is necessary. The reachability question *find ends in traces(jEdit,* m_{start}, m_{end} , ?) identifies a few statements (5 at a call depth of 5 or less from m_{start}) while a static slice from m_{start} finds all of the statements in hundreds of methods. Because the first line of *m* conditionally throws an exception depending on the input to *m*, everything afterwards is control dependent on the input to *m*. If this were not the case, the static slice still would not help locate *ends* and might not even include these statements if they do not happen to be control or data dependent. Even the searches supported by chopping are different: in chopping, both the origin and target statement are supplied by the user. Thus, the user must already know the statements in *ends* when they ask a *chop* question.

4. STUDY 1 – LAB OBSERVATIONS

In a previous study [12], we observed 13 developers at work on two 1.5 hour long changes to an unfamiliar codebase. We reported that experienced developers used their more extensive knowledge to diagnose the problem and formulate a fix addressing the underlying cause of the design problem rather than simply its symptoms [12]. Here we reanalyzed this study's data and report several new findings. Despite spending almost the entire task asking questions and investigating code, developers frequently incorrectly under-

False assumption or question related to a bug	Correct answer	Related reachability question	Dist	Notes
Method *m* is fast enough that it does not matter that it is called more frequently.	This method sends an event which triggers a hidden call to an extremely expensive library function.	*find ends in traces(jEdit,* m_{start}, m_{end} *, ?)*	4	Finds calls to downstream library functions in *m*
Why is calling *m* necessary?	*m* determines if the screen needs to be repainted and triggers it if necessary.	*find ends in traces(jEdit,* m_{start}, m_{end} *, ?)*	5	Finds calls to library functions, including one that triggers screen repainting
From what callers can the guards protecting statement *d* in method *m* be true?	More than one caller can reach *d.*	*find callers(m) in traces(jEdit, ?, d, ?)*	1	Finds callers reaching *d*
Method *m* need not invoke method *n* as it is only called in a situation in which *n* is already called. (2 bugs)	Method *m* is called in several additional situations.	*find callers(m) in traces(jEdit, ?, m, ?)*	1, 2	Finds callers reaching *m*
The scroll handler *a* does not need to notify *b*, because *b* is unrelated to scrolling.	Method *b* updates the screen to reflect updated scroll data signaled by *a*.	*find grep("scroll") in traces(jEdit,* a_{start}, a_{end}*, ?)*	1	Finds statements in *b* that reads scroll data updated when *a* occurs
Removing this call in *m* does not influence behavior downstream.	*m* no longer clears a flag, disabling functionality downstream	*compare(traces(*$jEdit_{old}$, m_{start} *, ?, ?), traces(*$jEdit_{new}$, m_{start} *, ?, ?)*	4	Finds differences in behavior resulting from the change, including downstream functionality that is no longer invoked.
What situations currently trigger this screen update in *m?*	A variety of user input events eventually cause *m* to be invoked	*find ends in traces(jEdit, ?, m, ?)*	3	Finds upstream methods with no callers, including user input event handlers called only by the framework.

Table 2. Questions developers failed to answer or false assumptions developers made in study 1 that are (1) associated with an implemented change containing a defect and are (2) associated with a reachability question. For each reachability question, Dist is the shortest call graph distance between the origin statement developers investigated and any statement found by the reachability question.

stood facts about the code. Acting on these false facts, developers implemented buggy changes, which in some cases they later realized were mistaken and abandoned. When developers inserted defects, we analyzed questions developers asked and actions they took to look for specific information they incorrectly understood. In other cases, developers spent tens of minutes employing tedious strategies. We report several of these strategies and questions they attempted to answer.

4.1 Method

We review the most important aspects of the method here. Additional details can be found in [12]. Participants were provided with the Eclipse 3.2.0 IDE and were allowed to use any Eclipse feature and take notes with Windows Notepad or on paper. Participants worked on two code-change tasks for 1.5 hours per task. Both tasks were changes to jEdit, an open source text editor that is 54,720 non-comment, non-blank lines of Java. Both tasks were designed to be challenging and require understanding the design of the code rather than just locating features or reusing an API. To achieve these goals, we designed both tasks to require fixing design problems. We searched the current version of the application for "HACK" comments and selected two problems. Both tasks involved editing code that controlled when updates happened and involved reasoning about related functionality scattered across the codebase. Participants found the tasks to be highly challenging – one participant described them as typical of a "bad day".

We conducted two new analyses of this data. First, we identified edits to the code and clustered these into changes. We labeled each change as to if it had been implemented, if it was later abandoned, and if it contained a bug. For changes containing a bug, we then looked to see if the developer had either asked a question or had otherwise made an assumption. We then attempted to determine if the question or assumption could be addressed by a reachability question. In a second analysis, we looked for examples of time-consuming questions that developers spent ten or more minutes answering.

4.2 Results

Developers implemented an average of 1.2 changes per task. Developers abandoned changes when they learned their changes could never work, found a bug they could not fix, or decided they did not have sufficient time to finish the change. Developers abandoned an average of 0.3 changes per task, two thirds of which contained bugs. Developers abandoned changes that did not contain a bug either because they no longer thought the change was a good design or did not think they had time to finish it. Overall, developers spent over two-thirds of their time (68%) investigating code – either testing or doing dynamic investigation using the debugger (22%) or reading, statically following call relationships, or using other source browsing tools (46%). They spent the remainder of their time editing (14%), consulting or creating other artifacts (task description, notes in Notepad, diagrams)(6%), or reasoning without interacting with any artifacts (11%).

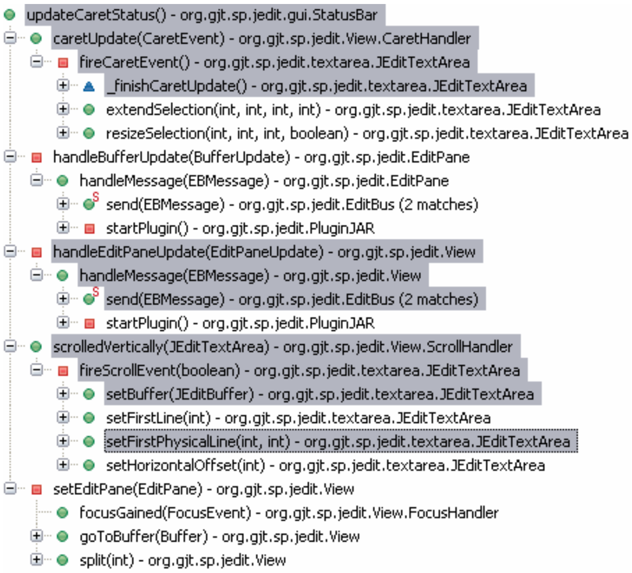
4.2.1 Causes of defective changes

Half of all changes developers implemented contained a bug. In half of these defective changes (8 changes), we were able to relate the bug to a reachability question either in a false assumption that developers made (75%) or a question they explicitly asked (25%). Table 2 lists the false assumptions or questions that were related to reachability questions and the corresponding reachability question. Developers often made incorrect assumptions about upstream or downstream behaviors as they reasoned about the implications of removing calls currently present in the code. These assumptions took different forms depending on the change they considered. *upstream* often occurred when developers asked or assumed that behavior was redundant and unnecessary because it would always be called somewhere else. In these cases, the call graph distance from the origin statement they were investigating to target behavior was often small (mean = 1.75). These questions were challenging to reason about because it was difficult to determine which calls were feasible. In contrast, *downstream* often occurred when developers made false assumptions about how a method mutated data or invoked library calls. Here, the relevant effect was further away (mean = 3.5 calls), and developers had no reason to believe that traversing the path to the target would challenge their assumption.

4.2.2 Tedious and time consuming strategies

In addition to the bugs that arose from assumptions developers made when they *should have* asked reachability questions, there were many cases where the developers *did* ask reachability questions and formulated a strategy to answer them. Developers spent much of the task investigating code by traversing calls in an attempt to understand what methods did and the situations in which they were invoked. Most participants rapidly switched between a call graph view (static) and the debugger call stack (dynamic). Static investigation allowed developers to navigate to any caller or callee at will. But as developers traversed longer paths of calls, developers were likely to hit infeasible paths. Several guessed incorrectly about which paths were feasible. Dynamic investigation was more time-consuming to begin – developers set breakpoints, invoked application behavior, and stepped through breakpoint hits until the correct one was reached. At task start, most investigation was relatively unfocused – developers attempted to make sense of what the methods did and the situations in which they were called. As the tasks progressed and developers began to propose changes, the questions grew increasingly focused and developers sought to navigate to specific points in code.

Developers differed greatly in the effectiveness and sophistication of the strategies they employed. Particularly challenging for many participants was upstream navigation. Two participants did not realize they could search the call stack to find an upstream method and instead spent much time (16 mins, 10 mins) locating the method by using string searches and browsing files. Three participants spent ten or more minutes (17, 13, and 10 mins) using a particularly tedious strategy to navigate upstream from a method *m* across only feasible paths: adding a breakpoint to each of m's callers, running the application, executing functionality, noting which callers executed, and recursing on these callers. Many participants used Eclipse's call graph exploration tool to traverse calls, but both traversed infeasible paths and experienced problems determining which calls led to their search targets (figure 1). The three most experienced participants instead invoked functionality and copied the entire call stack into a text editor. But even these experienced participants experienced problems reasoning about reachability relationships. Three of the defects inserted associated with reachability questions were inserted by these participants.



```
● updateCaretStatus() - org.gjt.sp.jedit.gui.StatusBar
⊟ ● caretUpdate(CaretEvent) - org.gjt.sp.jedit.View.CaretHandler
    ⊟ ■ fireCaretEvent() - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ▲ _finishCaretUpdate() - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● extendSelection(int, int, int, int) - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● resizeSelection(int, int, int, boolean) - org.gjt.sp.jedit.textarea.JEditTextArea
⊟ ■ handleBufferUpdate(BufferUpdate) - org.gjt.sp.jedit.EditPane
    ⊟ ● handleMessage(EBMessage) - org.gjt.sp.jedit.EditPane
        ⊞ ●S send(EBMessage) - org.gjt.sp.jedit.EditBus (2 matches)
        ⊞ ■ startPlugin() - org.gjt.sp.jedit.PluginJAR
⊟ ■ handleEditPaneUpdate(EditPaneUpdate) - org.gjt.sp.jedit.View
    ⊟ ● handleMessage(EBMessage) - org.gjt.sp.jedit.View
        ⊞ ●S send(EBMessage) - org.gjt.sp.jedit.EditBus (2 matches)
        ⊞ ■ startPlugin() - org.gjt.sp.jedit.PluginJAR
⊟ ● scrolledVertically(JEditTextArea) - org.gjt.sp.jedit.View.ScrollHandler
    ⊟ ■ fireScrollEvent(boolean) - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● setBuffer(JEditBuffer) - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● setFirstLine(int) - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● setFirstPhysicalLine(int, int) - org.gjt.sp.jedit.textarea.JEditTextArea
        ⊞ ● setHorizontalOffset(int) - org.gjt.sp.jedit.textarea.JEditTextArea
⊟ ■ setEditPane(EditPane) - org.gjt.sp.jedit.View
      ● focusGained(FocusEvent) - org.gjt.sp.jedit.View.FocusHandler
    ⊞ ● goToBuffer(Buffer) - org.gjt.sp.jedit.View
    ⊞ ● split(int) - org.gjt.sp.jedit.View
```

Figure 1. Developers using Eclipse's call graph exploration tool to traverse callers found it difficult both to identify feasible paths and those leading to their target. These methods are shaded, but the actual target is several levels further away behind several methods with high branching factors.

4.3 Discussion

Despite spending much of the task investigating code, developers were often unsuccessful in correctly understanding what it did. Developers made many false assumptions about relationships between behaviors that in some cases led to defects. Developers' tools were ill-suited for answering reachability questions, often forcing them to use tedious and time-consuming strategies to answer specific well-defined questions. And had developers been able to more easily check their erroneous assumptions that led to defects, their changes might have been more accurate.

While these results suggest that reasoning about reachability relationships is important for developers understanding unfamiliar, poorly designed code, these results might not be generalizable. While we expect developers do work with such code in the field, it is unclear how typical such a task is. While the carefully controlled setting of a lab study allowed us to evaluate the success and accuracy to a degree impossible in the field, lab studies are never able to perfectly replicate conditions in the field. Understanding real code in more typical tasks might involve fewer and less challenging reachability questions. Developers working in the same codebase over a period of time might be able to use their knowledge to directly answer reachability questions as studies suggest developers learn facts including callers and callees of methods with increasing experience [V]. Developers had limited time in which to work, which likely led them to rush changes with less investigation than they might otherwise have done. And several developers did not seem to have had much experience understanding large, complex codebases. Are reachability questions frequent and challenging for developers at work in the field?

5. STUDY 2 – SURVEY

In order to understand the frequency and difficulty of reachability questions in the field, we conducted a survey of developers in which they rated 12 questions for difficulty and frequency.

5.1 Method

We randomly sampled 2000 participants from among all employees at Microsoft's Redmond campus listed as a developer in the address book. Each was sent an email inviting them to participate

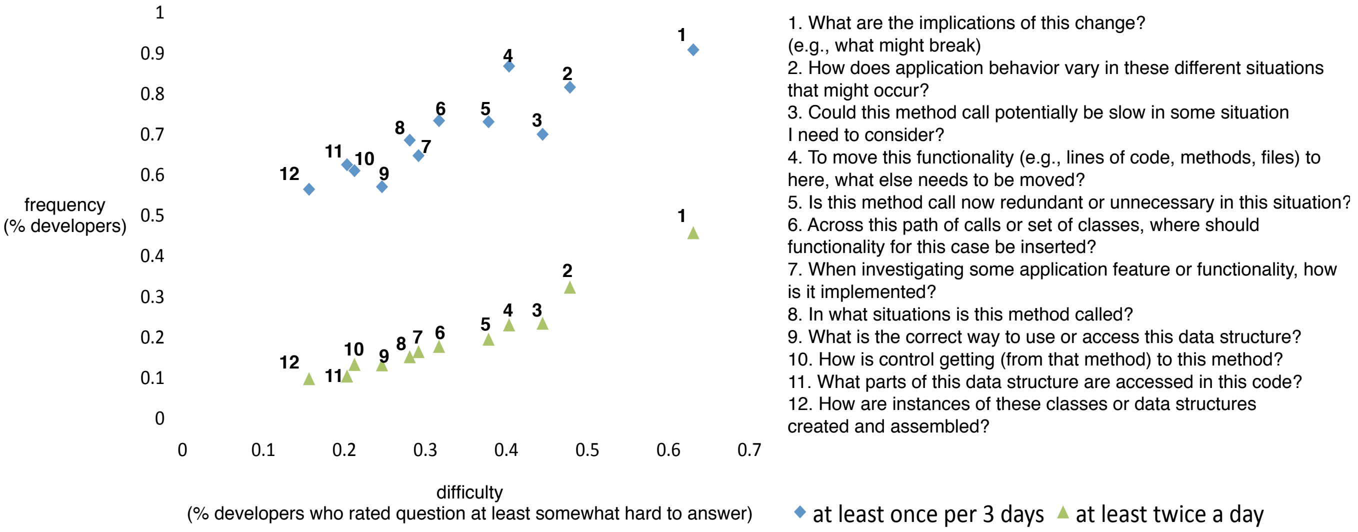


Figure 2. Frequency vs. difficulty for 12 reachability-related questions sorted by decreasing difficulty.

in our survey. We received 460 responses from developers and excluded 8 additional responses from non-developer positions. Respondents included 14 architects, 43 lead developers, and 403 developers. Most worked in a single or shared office while a small number (33) worked in an open, shared space. Respondents ranged in professional software development experience from the very inexperienced (0 years) to the very experienced (39 years), with a median of 9 years experience. Respondents frequently changed codebases, ranging in time spent in their current codebase from 0 to 8.33 years, but with a median of only 1 year. Nevertheless, 69% agreed that they were "very familiar" with their current codebase. Developers' teams were involved in a wide range of activities – 43% bug fixing, 34% implementation, 16% planning, and 7% other. Developers reported that they typically spent 50% of their work time editing, understanding, or debugging code, with a range from 0 to 100%.

In the main portion of the survey, developers were asked to rate the frequency and difficulty of 12 questions. These questions were selected from a previous study of questions that developers ask about code [17] and questions identified in our first study. Some of these were closely related to reachability questions ("In what situations is this method called?") while others were more indirectly related ("What are the implications of this change?"). However, we observed many of the indirectly related questions being refined into reachability question in our lab study. So, we hypothesized that developers often answer these questions by asking reachability questions.

We piloted the survey with 4 graduate students and 1 developer to ensure that the meaning of the questions was clear, and we iterated the wording based on the feedback. For each question, respondents were asked to rate how often in the past 3 days of programming they had asked the question and to rate its difficulty on a 7 point scale from very hard to very easy. 56 participants did not answer all questions. When a participant did not answer the questions necessary for a particular comparison, that participant was dropped from that comparison. To analyze the data, we looked both at simple descriptive statistics and correlations between ratings and demographic variables. We report these results using r (the Pearson product-moment correlation coefficient) and p (a statistical significance measure – smaller is more significant).

5.2 Results

On average, developers reported asking more than 9 of these questions every day. These questions were often hard to answer. Of the 12 questions that the developers rated, developers rated an average of 4.1 questions at least somewhat hard to answer and 1.9 as hard or very hard to answer. Few developers thought all these questions were easy to answer: 82% of respondents rated at least 1 question at least somewhat hard to answer, and 29% rated at least 1 question as very hard to answer. Surprisingly, developers do not ask these questions significantly less frequently and they are not significantly easier to answer as they become more experienced (r = -.07, p = .14; r = -.01, p = .81) or after spending more time in a codebase (r = -.04, p = .41; r = -.07, p = .15). Nor does the quality of the codebase significantly affect the frequency of these questions (r = -.08, p = .10). While it is harder to answer these questions on lower quality code (r = .36, p < .0001), it is not possible to say if this is unique to these questions or simply that all questions become harder to answer in poorly maintained code.

Figure 2 plots the questions' frequency against difficulty. Interestingly, difficulty was positively related to frequency (r = .35, p < .0001). Both the most frequent and hardest to answer question was "What are the implications of this change?" Generally, the most frequent and difficult questions were the most high level. For example, half of respondents reported asking "What are the implications of this change?" at least twice a day, and 63% of respondents rated it at least somewhat difficult to answer. Of course, some questions are much more frequent and difficult than others. Over 60% of developers thought answering "What are the implications of this change?" was usually at least somewhat hard to answer, while this was true of only 16% of respondents for "How are instances of these classes or data structures created and assembled?"

5.3 Discussion

Our results revealed that developers frequently ask questions that they might refine into reachability questions, that these questions are often difficult to answer, and that experience does not remove the need to ask these questions. These results suggest that answering these questions is an important part of how all developers understand code, whether they are new to a codebase or know it well and whether the codebase is poorly designed or well designed. These findings are still limited in that all survey respondents were taken from a single company. But respondents differed greatly by the products on which they worked, by experience with the codebase, by overall professional experience, and by software project phase. These results demonstrate that techniques that help developers more effectively answer these questions are important. However, the results do not establish that developers answer these questions by asking reachability questions. Do developers frequently ask reachability questions, and are they time consuming to answer? What do examples of reachability questions in the field look like?

6. STUDY 3 – FIELD OBSERVATIONS

In order to better understand the situations in which developers ask reachability questions and the strategies they use to answer them, we observed 17 developers at work on their everyday coding tasks.

6.1 Method

We recruited 20 developers at Microsoft from respondents to study 2 to participate in observation sessions. All sessions were conducted with a single observer and a single developer in the developer's office. Developers used a variety of programming languages (C++, C#, JavaScript), editors, and debuggers. After briefly introducing the observer and reviewing the purpose of our study, participants were asked to work on a coding task in their codebase for the remainder of the approximately 90 minute sessions. Three participants finished their first task and chose a second task. When selecting tasks, participants were encouraged to choose a task involving unfamiliar code, minimally defined as code they had not written themselves. While only 35% of the tasks that developers chose were tasks they planned to do at the time of our session, 95% (all but one) of the tasks they chose were on their lists of tasks to do. The remaining task was a bug previously assigned to another team-member. The work we observed was not biased towards the beginning or the end of tasks: 45% of the tasks were tasks the developer had previously begun, and developers completed 45% of their tasks. All but one developer stopped working after they had completed testing their fix and before having their teammates code-review the change.

We asked participants to think aloud as they worked. When deeply engrossed in the tasks, participants occasionally forgot to talk, and we prompted them to resume by asking what they were trying to do or having them confirm or reject a statement about what they appeared to be doing. To record the sessions, we recorded audio and took notes. Two of the recordings were lost due to equipment failure, leaving 18 participants. From the recordings and observer notes, we produced time stamped, annotated transcripts of the sessions spanning 386 pages.

To analyze the data, we first reviewed the transcripts and qualitatively summarized what developers were doing. Next, we iteratively designed a coding scheme for describing developers' activities. We coded 17 of the 18 sessions – one session did not include any implementation task. Each session was coded for activity at one-minute time granularity. Participants occasionally retrospectively described particularly memorable past tasks or talked about how they approached tasks in general which we did not include in the activity list, but mention in the discussion section. Developers were interrupted by replying to task-unrelated emails, by teammates dropping by, or discussions with the interviewer. All task-irrelevant activity was coded as an interruption and excluded from the analysis of time use. Due to equipment failure, we lost 15 minutes of the recordings out of a total of 962 minutes of task-related activity. In most cases, developers stopped working on their tasks once they had completed its implementation. But one developer reached the end of his task and conducted a code review. So we do not include code reviews in our activity times.

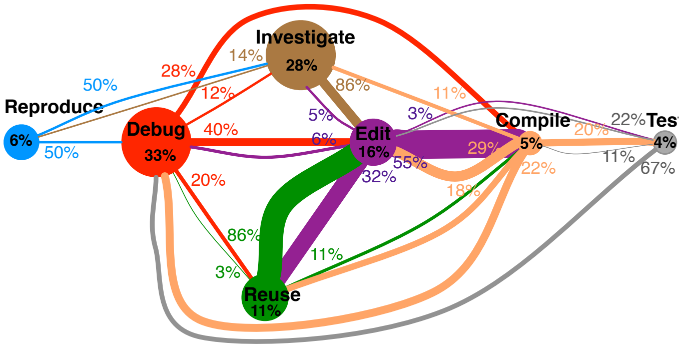


Figure 3. Developers' activities (circles with % of activity time) and transitions between activities (lines with % of transitions from activitiy). Transitions from an activity are in the activity's color, and left to right transitions are above right to left.

6.2 Results

Developers spent a majority of their time understanding code by debugging (33%) or proposing changes and investigating their implications (28%). 9 of the 10 longest debugging and implication investigations were associated with a reachability question.

6.2.1 Activities

Figure 3 depicts the sequence of activities we observed and the time developers spent on each. When working on a bug they did not already understand, developers first sought to *reproduce* the problem by following steps in the bug to confirm that the bug had not already been fixed, ensure that a fix could be tested, and provide a way to begin using the debugger. Developers faced with incorrect application behavior, either from the original bug or introduced by their fix, *debugged* to assign blame to specific program points exhibiting incorrect behavior. After determining the cause of a bug or when beginning a feature implementation task, developers began to propose fixes to solve the problem and *investigated* the implications of the proposals on program behavior. Developers then *edited* the code to implement the change. When editing, developers sometimes *reused* existing functionality and sought to learn its name and how to correctly reuse it. Developers *compiled* and built the application, sometimes producing compile errors they debugged. Finally, developers *tested* their changes, often revealing defects they debugged.

6.2.2 Time-consuming activities

While debugging and investigating code, developers frequently asked reachability questions. In order to examine the relationship of these activities to reachability questions, we looked for reachability questions in the 5 longest debugging and 5 longest investigation activities. Each of these activities had a central, *primary question* developers tried to answer throughout the activity. Surprisingly, the primary question in 9 out of 10 of these activities

Developer's primary question (Debugging activities)	Time (min)	Reachability question	Notes
Where is method *m* generating an error?	66	*find grep(errorText) in traces (p, m_{start}, m_{end}, ?)*	Finds the statement downstream from *m* outputting error text
What resources are being acquired to cause this deadlock?	51	*find ACQUIRE_METHODS in traces (p, o, d, ?)*	Finds calls to methods acquiring resources, including those leading to the deadlock.
"When they have this attribute, they must use it somewhere to generate the content, so where is it?"	35	*find reads(attribute) in traces(p, o, d, ?)*	Finds downstream uses of *attribute*, including those generating the content.
"What [is] the test doing which is different from what my app is doing?"	30	*compare(traces(p_{test}, o, d, ?), traces(p_{app}, o, d, ?))*	Finds differences in behavior between the test program and app program
How are these thread pools interacting?	19	*find methods(T) in traces(p, o, d, ?)*	Finds any calls into methods in thread pool types *T*.

Developer's primary question (Investigation activities)	Time (min)	Reachability question	Notes
How is data structure *struct* being mutated in this code (between *o* and *d*)?	83	*find writes(struct) in traces(p, o, d, ?)*	Finds all downstream statements mutating *struct*
"Where [is] the code assuming that the tables are already there?"	53	*compare(traces(p, o, d, tablesLoaded), traces(p, o, d, tablesNotLoaded))*	Finds different behaviors the code exhibits when tables are not loaded
"How [does] application state change when *m* is called denoting startup completion?"	50	*find writes(FIELDS) in traces(p, m_{start}, m_{end}, ?)*	Finds state changes caused by *m*
"Is [there] another reason why status could be non-zero?"	11	*find dDepend(status) in traces(p, ?, d, ?)*	Finds upstream statements through which values flow into *status*, including those creating its values

Table 3a (top) and 3b (bottom). The 5 of the 5 longest debugging activities and the 4 of the 5 longest investigation activities associated with a reachability question. For each activity, the developer's primary question during the activity, the length of the activity, and the related reachability question.

was a reachability question. At the beginning of these activities, developers rapidly formulated a specific question expressing search criteria describing statements they wished to located. For example, to debug a deadlock, a developer began at a statement and began traversing callees in search of statements acquiring resources. 51 minutes later, this finally revealed the sequence of behaviors causing the deadlock.

When answering reachability questions, developers explored the code either dynamically using the debugger and logging tools or statically using source browsing tools. Interestingly, developers did not primarily use the debugger to debug and code browsing tools to investigate implications. Instead, like the lab study participants, developers often made use of both tools as they sought to answer multiple lower-level questions or tried alternative strategies for answering their primary question. Developers constantly dealt with uncertainty during their tasks both from generating and testing hypotheses and wondering about the correctness of results produced by their tools.

An example from the longest debugging activity helps illustrate several of these points. Observing an error message in a running application, one developer spent 66 minutes locating the cause of the error message in the code. Using knowledge of the codebase, he rapidly located the code implementing the command he had invoked in the application. But it was not obvious where it triggered the error. Hoping to "get lucky", he did a string search for the error message but found no matches. Unsure why he did not find any matches, he next began statically traversing calls from the command method in search of the error. But he rapidly determined he was unsure which path would be followed when the command was invoked. Switching to the debugger, he stepped through the code until learning his project was misconfigured and creating spurious results both in his debugger and code searches. After resetting his project configuration, he again did a string search for the error string and found a match. However, many callers called the method, any one of which might be causing his error. So he returned to stepping in the debugger. Finally locating code that seemed relevant, he quickly browsed through the code statically. Finally, he returned to the debugger to inspect the values of some variables.

6.3 Discussion

In the third study, developers spent over half of their time debugging or reasoning about the implications of their changes. In 9 of the 10 most time-consuming activities, the developer's primary question was a reachability question. Developers were at a point in code and had specific search criteria describing the statements they wished to find. But finding these statements was hard and time consuming as developers searched through large amounts of task-irrelevant code. In contrast to results from study 1, the questions in study 3 were all questions developers explicitly asked.

Like all studies, these findings may have been influenced by the practices and tools that developers used that might differ in other organizations. In organizations with more extensive documentation or commenting processes, developers might rely on these more than the code itself. Developers did not have access to sophisticated UML reverse-engineering tools. None of our developers had unit tests extensive enough to rely on to test the correctness of their changes. Extensive unit tests might lead to more implementation of speculative changes, followed by testing, rather than extensive investigation prior to changes.

7. GENERAL DISCUSSION

We found that reachability questions are frequent, often hard to answer, associated with false assumptions that lead to bugs, and asked by developers in many of the most time consuming debugging and investigation tasks. Several developers in the lab study became so overwhelmed investigating code that they gave up. Developers at work on actual tasks in the field often spent tens of minutes answering single reachability questions when debugging or investigating the implications of their changes. In all of these cases, developers asked questions and explored the code to search for statements answering their questions. Linking many of the diverse problems developers commonly experience understanding large complex codebases to reachability questions helps better explain the strategies developers use to understand code and the factors influencing their success or failure.

7.1 Strategies for answering reachability questions

Developers may choose from among several classes of strategies for answering questions: reasoning using facts they already know, communicating with teammates, or dynamically or statically exploring code. For code that developers know well, developers may already know the answer [6]. But this level of understanding is difficult to achieve due both to the number of reachability relationships present in a codebase and because they often change as developers edit the code. One field study participant spent several minutes investigating code he had written himself a little over a year earlier because he was not certain of several important details unique to his task and he was concerned others might have edited the code. Conversely, even developers new to a codebase are able to generate hypotheses about reachability relationships by interpreting identifiers and using their knowledge about how they expect an application to work. Study 1 participants assumed that an EditBus was connected to edit events. But when developers wished to test these hypotheses, they used other strategies.

Developers communicate with their teammates both directly through face-to-face communication, instant message, or email and indirectly through documentation and comments. Where they exist, documentation diagrams such as UML sequence diagrams could help answer some reachability questions provided they anticipate the correct question. But nearly all of the questions we observed were highly specific to the developers' task making it unlikely for that such a diagram would exist. Developers occasionally made use of direct communication, often instant messaging teammates they thought might know all or part of an answer. But teammates often were not available to immediately respond. Moreover, for longer face-to-face interruptions, developers are sometimes expected to have already done due diligence to get a general understanding before asking a lengthy question of a busy and more knowledgeable teammate [13]. Of course, teammates also eventually leave the team, may be otherwise unavailable, might have forgotten the answer, or might never have known the answer at all.

Thus, developers often answered reachability questions by exploring the code. In dynamic exploration, developers run the program and observe its output either directly or through tools such as a breakpoint debugger, logging statements, or logging tools. In some cases, generating the trace to be dynamically investigated was difficult or impossible because special hardware was required, it took a long time for the application to run and generate the trace, or it was unclear what application input was necessary to generate the trace. A developer in study 3 working with a web application added logging statements before waiting a day for it to execute a lengthy batch job. Moreover, some reachability questions forced consideration of all possible traces. Developers sometimes randomly invoked application behavior in an attempt to generate desired traces. When possible, there were several advantages of dynamic exploration. Developers could inspect state and even mutate state to select the trace being followed. Breakpoints allowed developers to search for paths to a statement. But setting breakpoints was impractical when searching for many statements (e.g., any method in a type) or when developers did not know the statements for which they were searching (e.g., all statements related to scrolling).

Some of the problems we observed in the lab study could be attributed to a lack of knowledge of effective dynamic investigation strategies. Developers exploring upstream by iteratively setting breakpoints could have instead much more effectively inspected call stacks. However, developers devising and choosing strategies must simultaneously hypothesize answers to their questions, keep track of the question they are answering and information they have found, and deal with frequent interruptions from teammates [10]. In these situations, developers may not have time to reflect at length on their strategies. However, better educating developers about the types of questions they ask and the strategies they could use to answer them might help them devise more effective code exploration strategies.

7.2 Challenges statically exploring code

In static exploration, developers navigate the code by using source browsing tools such as a call graph exploration tool or textual searches for names. In contrast to dynamic exploration, static exploration does not require running the program. Call graph tools, such as the Eclipse call hierarchy, allow developers to follow chains of calls through the source. However, we observed many cases where these chains contained infeasible paths that could never execute. Infeasible paths are caused by correlated conditionals where the branch taken at a (consumer) conditional is correlated to one of several producers controlled by the path by which the consumer was reached. Through our direct observations and retospective accounts from our participants, we discovered several idioms that created correlated conditionals with widely separated producers and consumers that were particularly difficult to statically explore. In an event bus architecture, messages are created by a producer, sent over a bus, and subscribed to by consumers. In COM, a pointer is initialized to a particular implementation of an interface (producer) and passed to call sites invoking methods on the interface (consumer). In frameworks, clients often register their implementations of framework interfaces with the framework (producer) which then uses dynamic dispatch (consumer) to transfer control back. In a property system, values referring to properties are created (producer) and used to access property getters or setters which look up the property (consumer).

Several, but not all, of these idioms often produce high branching factors in the control flow graph. A common interface (e.g., IRunnable in Java) may have many implementations, creating a large branching factor at dynamic dispatch. In an event bus, many methods call the bus send method and many bus receive methods are called by the bus, creating two high branching factors. For the developer, the effect of correlated conditionals is to create many possible edges to traverse, forcing the developer to guess which are feasible or attempt to manually simulate control flow by propagating data over control flow paths. We observed that performing path simulation manually was nearly impossible for statements with high branching factors as there were simply too many paths to consider.

7.3 Recommendations for tools

We found that developers often had specific search criteria describing statements they wished to find. However, most existing tools force developers to guess where these statements might be located when they traverse calls statically or dynamically. Recently, a few tools have begun to support searching. In Dora [8], a developer specifies an origin method and a string search criteria, and Dora scores methods connected by a call graph path by their relevancy to the search string. However, Dora does not eliminate infeasible paths and supports only searches described by string (*grep(str)* in our formalism), not attributes of target statements. The OASIS Sequence Explorer [1] depicts a trace in a UML sequence diagram and allows developers to use a regular expression to search for method names. Our findings suggest improved tools could greatly improve developer productivity by supporting searches across feasible paths for statements matching a wider variety of search criteria.

8. CONCLUSIONS

Modern development environments provide developers with a debugger and source browsing tools for exploring code. But we found that these tools only indirectly answer many of the questions developers ask or should have asked. Better educating developers about reachability questions might help developers learn, share, and choose more effective strategies for answering reachability questions. But our results also suggest that developers could perform coding tasks more quickly and accurately with tools that more directly support answering reachability questions.

9. ACKNOWLEDGMENTS

We thank our participants for participating in our studies, and Jim Herbsleb for helping conduct study 1. Studies 2 and 3 were conducted while the first author was a visitor at the Human Interactions in Programming team in Microsoft Research. We thank Jonathan Aldrich for helpful discussions of our formalism. This research was funded in part by the National Science Foundation, under NSF grant CCF-0811610.

10. REFERENCES

[1] Bennett, C., Myers, D., Storey, M., German, D. M., Ouellet, D., Salois, M., and Charland, P. (2008). A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *J. Softw. Maint. Evol.*, 20 (4), 291-315.

[2] Cherubini, M., Venolia, G., and DeLine, R. (2007). Building an Ecologically valid, Large-scale Diagram to Help Developers Stay Oriented in Their Code. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*

[3] Détienne, F. (2002). *Software Design---Cognitive Aspects.* Springer-Verlag.

[4] Dzidek, W. J., Arisholm, E. and Briand, L.C. (2008). A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. In *TSE*, 34 (3).

[5] Edwards, J. (2009). Coherent reaction. In *Proc. of Onward!* at the *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 925-932.

[6] Fritz, T., Murphy, G. C., and Hill, E. (2007). Does a programmer's activity indicate knowledge of code? In *ESEC/FSE*.

[7] Harman, M. and Hierons, R. (2001). An overview of program slicing. In *Software Focus*, 2(3), 85-92.

[8] Hill, E., Pollock, L., and Vijay-Shanker, A.K. (2007). Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. of Automated Software Engineering (ASE).*

[9] Ko, A. J., Aung, H., and Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. Int'l Conf. Software Eng (ICSE),* 126-135.

[10] Ko., A.J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. In *Proc. Int'l Conf. Software Eng (ICSE).*

[11] Ko, A., Myers, B., Coblenz, M., and Aung, H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. In *IEEE Trans. Soft. Eng.* 32, 12 (2006), 971 - 987.

[12] LaToza, T.D., Garlan, D., Herbsleb, J., and Myers, B.A. Program Comprehension as Fact Finding. *In European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2007.

[13] LaToza, T.D., Venolia, G., and DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. Int'l Conf. Software Eng (ICSE)*, 492-501.

[14] Lawrance, J., Bellamy, R., Burnett, M. and Rector, K. (2008). Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. In *Proc. of CHI*, 1323-1332.

[15] Parent, S. (2006). A possible future for software development. Keynote talk at the *Workshop of Library-Centric Software Design, OOPSLA.*

[16] Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. In *Cognitive Psychology*, Vol. 19, 295-341.

[17] Sillito, J., Murphy, G.C., and De Volder, K. (2008). Asking and answering questions during a programming change task. *TSE*, 34(4).

[18] Singer, J., Lethbridge, T., Vinson, N., Anquetil, N. (1997). An examination of software engineering work practices. In *Proc. CASCON*, 209-223.

[19] Sridharan, M., Fink, S. J., and Bodik, R. (2007). Thin slicing. In *Programming Language Design and Implementation (PLDI).*

[20] Tip, F. (1995). A survey of program slicing techniques. In *Journal of Programming Languages, 3,* 121-189.

[21] Weiser, M. (1982). Programmers use slices when debugging. In *Communications of the ACM (CACM)*, 25(7), 446-452.

[22] Weiser, M. (1984). Program Slicing. In *Transactions on Software Engineering (TSE)*, 10 (4).