Information Needs in Collocated Software Development Teams

Amy J. Ko
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213
ajko@cs.cmu.edu

Robert DeLine and Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{rdeline, ginav}@microsoft.com

Abstract

Previous research has documented the fragmented nature of software development work. To explain this in more detail, we analyzed software developers' day-to-day information needs. We observed seventeen developers at a large software company and transcribed their activities in 90-minute sessions. We analyzed these logs for the information that developers sought, the sources that they used, and the situations that prevented information from being acquired. We identified twenty-one information types and cataloged the outcome and source when each type of information was sought. The most frequently sought information included awareness about artifacts and coworkers. The most often deferred searches included knowledge about design and program behavior, such as why code was written a particular way, what a program was supposed to do, and the cause of a program state. Developers often had to defer tasks because the only source of knowledge was unavailable coworkers.

1. Introduction

Software development is an expensive and time-intensive endeavor. Projects ship late and buggy, despite developers' best efforts, and what seem like simple projects become difficult and intractable [2]. Given the complex work involved, this should not be surprising. Designing software with a consistent vision requires the consensus of many people, developers exert great efforts at understanding a system's dependencies and behaviors [11], and bugs can arise from large chasms between the cause and the symptom, often making tools inapplicable [6].

One approach to understanding why these activities are so difficult is to understand them from an information perspective. Some studies have investigated information sources, such as people [13], code repositories [5], and bug reports [16]. Others have studied means of acquiring information, such as email, instant messages (IM), and informal conversations [16]. Studies have even characterized developers' strategies [9], for example, how they decide whom to ask for help.

While these studies provide several concrete insights about aspects of software development work, we still know little about what information developers look for and why they look for it. For example, what information do developers use to triage bugs? What knowledge do developers seek from their coworkers? What are developers looking for when they search source code or use a debugger? By identifying the types of information that developers seek, we might better understand what tools, processes and practices could help them more easily find such information.

To understand these information needs in more detail, we performed a two-month field study of software developers at Microsoft. We took a broad look, observing 17 groups across the corporation, focusing on three specific questions:

- What information do software developers' seek?
- Where do developers seek this information?
- What prevents them from finding information?

In our observations, we found several information needs. The most difficult to satisfy were design questions: for example, developers needed to know the intent behind existing code and code yet to be written. Other information seeking was deferred because the coworkers who had the knowledge were unavailable. Some information was nearly impossible to find, like bug reproduction steps and the root causes of failures.

In this paper, we discuss prior field studies of software development, and then describe our study's methodology. We then discuss the information needs that we identified in both qualitative and quantitative terms. We then discuss our findings' implications on software design and engineering.

2. Related Work

Several previous studies have documented the social nature of development work. Perry, Staudenmayer and Votta reported that over half of developers' time was spent interacting with coworkers [15]. Much of this communication is to maintain awareness. De Souza, Redmiles, Penix and Sierhuis found that developers send emails before check-ins to allow their peers to prepare for

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

changes [5]. Collocation is a central factor in determining the quality of awareness information. Seaman and Basili found that local mobility facilitates awareness in ways that are unavailable in distributed situations [18]. Similarly, coordination problems can be exaggerated across sites because of the lack of spontaneous communication channels [8].

Developers also communicate to obtain knowledge [9]. LaToza, Venolia and DeLine describe the role of the "team historian," who possesses knowledge about the origins of a project and its architecture [13]. To determine who to ask, developers often gauge expertise by inspecting check-in logs [5], but such information is not always accurate [12].

One consequence of developers' frequent communication is the fragmentation of time. Gonzalez, Mark and Harris found that developers average about 3 minutes on a task and about 12 minutes in an area of work before switching [7]. These switches occur due to changing task priorities and getting blocked [15]. Perlow related how one software group's frequent interruptions created a sense of a "time famine"—having too much to do and not enough time [14].

Dependencies are also a central factor in software development. Developers use bug reports, content management systems, and version control systems to manage dependencies and notify coworkers of new dependencies [5]. Teams will clone software to avoid dependencies, even though they later have to duplicate fixes to the cloned code [13]. Developers also rush their activities to minimize dependencies between their code and recently committed changes in the repository [5].

These previous studies provide a general sense of the importance of communication among developers to maintain awareness, share knowledge, and manage dependencies. Using similar methods, both this study and a recent study by Sillito, Murphy and De Volder [1] dissect this communication from an information needs perspective by cataloging the questions that arise during development tasks. Sillito, Murphy and De Volder present 44 questions about code that developers asked during programming tasks. In this study, we present 21 questions that developers asked during their daily work (designing, coding, debugging, bug triage). Our 21 questions cover a broader scope of work and are therefore more abstract than their 44 questions. Unifying these results is future work.

9:41 am	So this copies the files onto the server, then allocates a machine to do the setup. In the meantime, I'm going to get this local fix [of this other bug] over [checked in].
9:41 am	[opens diff tool to see changes he's made to code]
9:43 am	Oh damn, I have some mixed changes. Some are part of this DCR [design change request] I'm working on and some are part of a bug fix, so I have to mix it out.

Figure 1. An excerpt from J's observation log.

3. Method

Our method was to record notes while observing developers' normal work. To recruit developers, we surveyed 250 developers from the corporate address book. Of these, 55 responded and 49 volunteered for observation.

Each observation session was about 90 minutes and involved a single observer taking handwritten notes. To encourage the participant to narrate his work, we asked the participant to think of us as a newcomer to the team, doing a "job shadow." We focused on recording goal-oriented events like "finding the method that computed the wrong value" rather than low-level events like keystrokes or menu selections. Since we shared the participants programming background, we understood much of the work and where and how information was obtained, without inquiry. In some cases, we prompted with questions like "what are you looking for?" to learn their information needs, but most developers thought aloud without prompting. We timestamped the recorded events and conversations each minute. After 90 minutes, we looked for a good stopping point and wrapped up. Immediately after each observation, we transcribed the handwritten notes, as in the excerpt shown in Figure 1.

During the allotted time for the study we were able to observe 17 developers, which was enough to see common patterns in their information needs. (Section 6.2 touches on the potential value of observing more developers.) Figure 2 describes these developers' experience levels, types of work, and phases of development and introduces the initials we use to refer to them in this paper. In Microsoft's terminology, *dev leads* manage software *development engineers (SDEs or devs)* while also performing a development role.

4. Task Structure

Our observations spanned 25 hours of work. We partitioned the logged activities into work categories common across the participants: writing code; submitting code (check-ins); triaging bugs; reproducing failures; understanding program behavior; reasoning about design; maintaining awareness; and non-work activities (e.g. personal phone calls). We also identified causes of task switching: face-to-face dialogue; phone calls; instant messages (IM); email alerts; meetings; task avoidance; getting blocked; and task completion. We annotated the logs

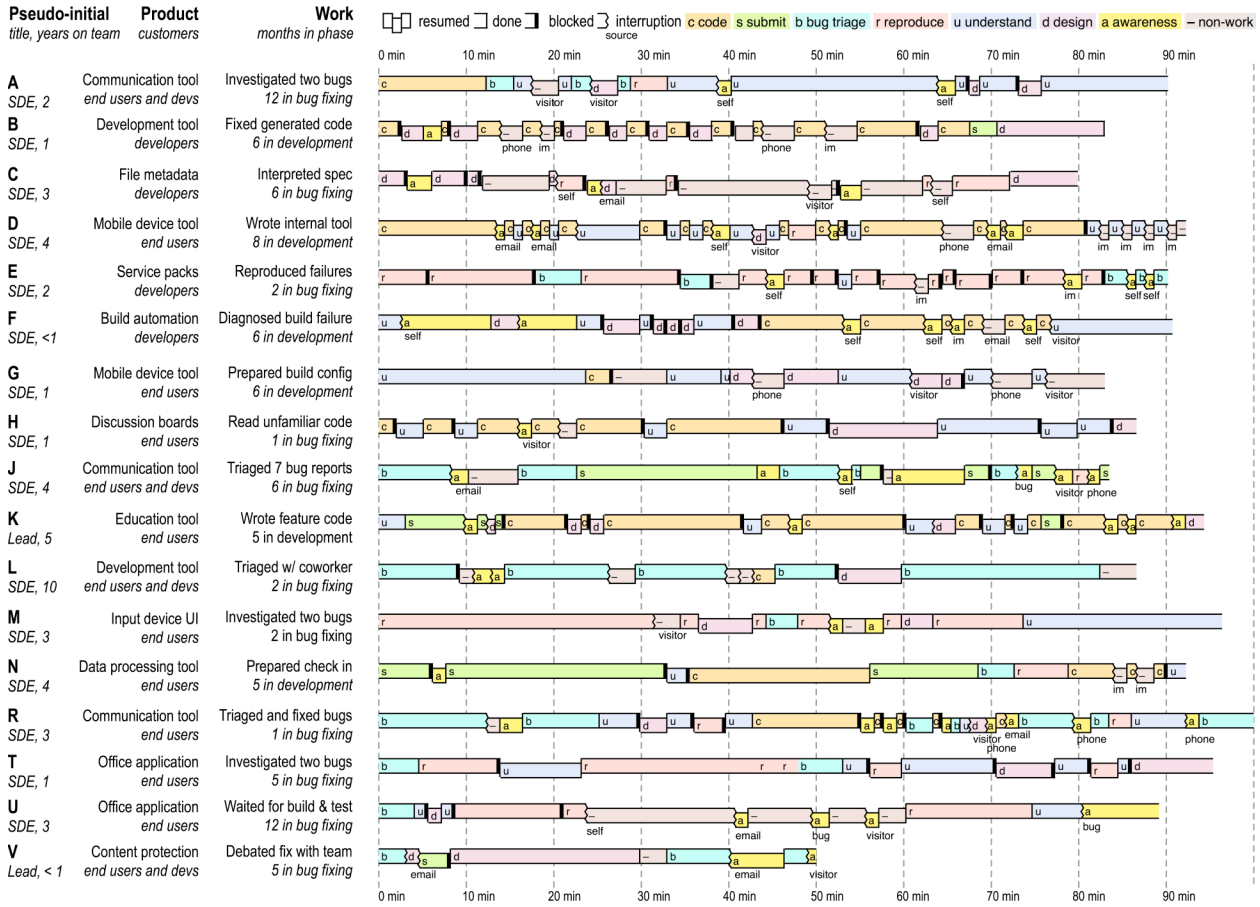

Figure 2. The backgrounds and task structures of the 17 observed developers. The right edge of each task block indicates the reason for the task switch (thin line for *done*, thick line for *blocked*, jagged line for *interrupted*). When a task gets broken up by interruptions or blocking, we draw its fragments at the same vertical level to show resumption.

with these switches, based on remarks like "I want to get back to my repro…" All of these causes of task switches are forms of interruption, except getting blocked and task completion. When the participant voluntarily switched activities, we label the switch as *blocked* if the participant could no longer make progress on the current activity (typically due to an information need) and *task avoidance* if she could make progress but chose to switch anyway.

Figure 2 visualizes these task switches, which occurred an average of every 5 minutes (± 1.7), mirroring the rate reported in Gonzales, Mark and Harris [7]. Time fragmentation varied considerably per participant. For example, M reproduced failures without interruption, whereas R was frequently blocked. Many interruptions were due to face-to-face, IM, or phone conversations, which occurred from 0 to 6 times per session (median of 1), each lasting for much of the session. Developers were also interrupted by notifications, such as email and alerts about changes to the bug database. Developers experienced from 0 to 9 notifications per session (median of 1).

Blocking, shown in Figure 2 as dark vertical bars, occurred when information was unavailable. Some blocks were about waiting for the results of compilations and tests suites. Developers also waited for email replies and for other teams to submit changes or x bugs. Other blocks were due to missing knowledge, like when a developer stops coding to learn about an API. Developers were blocked a median of 4 times per session and between 0 and 11 times overall.

5. Information Needs

During our observations, there were 334 instances of information seeking, which we abstracted from the particulars of the work context into 21 general information needs. Here we present these information needs clustered by the work category in which they arose. Throughout,

we list within braces the initials of the developers for whom we observed an information need or other trend.

5.1 Writing Code

Developers had several questions situated in the code they were writing:

(*c*1) *What data structures or functions can be used to implement this behavior?*
(*c*2) *How do I use this data structure of function?*
(*c*3) *How can I coordinate this code with this other data structure or function?*

Although the first of these questions was uncommon, when it occurred, developers searched API documentation {K} and inspected other code for examples {H}. These searches can be thought of as a search through the space of existing reusable code; for example, K looked for an appropriate serialization API by searching a large database of public documentation.

Once a developer had a candidate in mind, they sought its syntactic usage rules (*c*2). For example, which method is appropriate to call? What data structures does this require? What constructors does this class have? Developers used documentation when it was available {DFHJKN}, but sometimes needed to use code that was only fully understood by its author {AFL}. Others found example code from which to infer rules {GH}.

Because developers had to coordinate APIs with their own code, they also sought behavioral usage rules (*c*3), implicit in the API design. For example, is it legal to call this method after calling this other method? What state do I have to be in before this call? Such information was rarely explicit. Developers used their colleagues {A}, documentation {K}, and example code {HN} to infer these rules.

5.2 Submitting a Change

Developers had three primary questions when exposing their code to their teammates:

(*s*1) *Did I make any mistakes in my new code?*
(*s*2) *Did I follow my team's conventions?*
(*s*3) *Which changes are part of this submission?*

Besides building their code to assess its syntactic correctness, developers answered questions of correctness (*s*1) by considering the scenarios and range of input that they intended to cover. They used debuggers {DKR}, diff tools {R} and unit tests {BN}, but primarily relied on their own reasoning {ADHJ}. Another common filter for mistakes was code reviews. Before a review, developers first looked for mistakes:

> K: I think I'm ready to check in, so I'm just making sure I didn't do anything stupid. Like, I forgot to write those tests! Yeah, stupid like that.

One developer wrote assertions {N}, but these interfered with other developers' work {ATU}:

> A: I never want to see [that product's] asserts but they always pop up. They have nothing to do with my work!

Three developers used static analysis tools to check for fault-prone design patterns {JKU}, but expressed disdain for such tools' false positives or could not understand the tools' recommendations.

Developers also considered their team's conventions (*s*2). Some teams required tags or other documentation to be embedded in method headers, which developers were careful to remember, often with the help of tools {BELN}. Sometimes two submissions intersected (like in the transcript in Figure 1) or developers had to merge their code with another's and developers had to determine which differences were part of the current submission (*s*3) {JN}.

5.3 Triaging Bugs

Most developers were swamped with bug reports from tests, customers, and internal employees. Triage occurred in isolation as a developer partitioned their time {AEJMNTUV}, but also in triage meetings {LR}. For each report, the goal was to determine:

(*b*1) *Is this a legitimate problem?*
(*b*2) *How difficult will this problem be to fix?*
(*b*3) *Is the problem worth fixing?*

Assessing legitimacy (*b*1) involved deciding whether a failure was due to a problem with the code or an unrealistic configuration of a test {BL}:

> B: It might but not really be a failure. It might just be a setup problem. This particular component doesn't depend on anything. Probably locked a file, so it's returning an exit code. Not a real failure.

Legitimacy also depended on whether a report was a duplicate. People reported similar failure symptoms {L}, but also different failure symptoms that developers believed had a common cause {LT}:

> A: These subjects are just busted! I have a feeling I'm seeing the same bug. I'm going to do a quick search to see if there are busted subjects [in the bug database]—this one kinda sounds like it, blah blah, category name is corrupted? Ooh, screenshots are the same!

Bug triage is a cost/benefit analysis. To assess a the cost of repair (*b*2), developers considered whether a redesign would be necessary {CJTV}, whether other teams might be affected {V}, and whether a fix could be written and tested by a deadline {V}. Teams also close bugs "by design," treating them as work items for later releases:

> V's teammate: I think the best thing is a new overlay to indicate something's going on.

V: We can't do that by the release. Looks like a work item.

Another factor affecting the repair cost was a reports' clarity {JLR}. Does it have detailed reproduction steps? Is the failure clearly described? Does it have hints about possible causes or an error message? Reports with inadequate clarity were rejected {R}.

On the benefit side of the analysis ($b3$), developers considered the number of users affected {CJLV} and the user experience {LRV}. For example, V discussed a fix:

V's teammate: If we want to push it back, we can, but I think an overlay is easiest.

V: But it's a totally broken experience for the user.

If there was a known workaround, developers might focus on more severe bugs {LV}.

5.4 Reproducing a Failure

To reproduce a failure, developers asked:

($r1$) *What does the failure look like?*
($r2$) *In what situations does this failure occur?*

The primary source for both of these types of information was bug reports. Reports would often include screen shots {MT}, but more often developers relied on the descriptions of the failure to help them imagine its appearance {AELNRTU}.

Developers relied heavily on bug report's reproduction steps to understand the situations in which a failure occurred ($r2$). Given the complex configurations that were necessary to reproduce some problems, even detailed steps omitted crucial state {ERT}. In other cases, the state was known, but difficult to reproduce {AT}:

A: Originally, the repro steps said I need a blog count [as a test case] but I couldn't set one up, so I went back and forth.

To overcome this, some developers set up a remote desktop connection with the report's author, so that the full configuration was available for debugging {EL}. Developers would also guess what state was wrong and begin modifying their environment and test cases until reproducing the failure:

A: I'm looking at [the report] to see if I have this configured the same way, but I'm not getting the problem. Maybe we've changed it in the past half year this has been open.

In one situation a failure could not be reproduced and the bug had to be deferred {A}. The developer documented his attempts in the report for the sake of other testers and developers.

5.5 Understanding Execution Behavior

Developers had to understand unfamiliar code in several circumstances: using vendor code {GM}; joining a new team {V}; obtaining ownership of code {H}; during workload balancing {T}; or when debugging, with unfamiliar code on the call stack {ENT}. Each time, they addressed three basic questions:

($u1$) *What code could have caused this behavior?*
($u2$) *What's statically related to this code?*
($u3$) *What code caused this program state?*

Developers began these tasks with a why question and a hypothesis about the cause of the failure:

A: Why did I get gibberish? Storing field, given PPack, what is an MPField? I have no idea what this data structure contains. SPSField? I suspect SPS is just busted.

Developers acquired their hypotheses ($u1$) by using their intuition {ALM}, asking coworkers for opinions {AFM}, looking execution logs {F}, scouring bug reports for hints {ER}, and using the debugger {GTU}. Although developers used many sources to obtain hypotheses, only a few gathered and considered more than one at a time {FM}. The accuracy of developers' hypotheses was only obvious in hindsight.

To test and refine hypotheses, developers asked a broad array of questions with a variety of tools. Many of these questions were about the structure of the code ($u2$), like *what is the definition of this?* and *what calls this method?* Such questions were easy to answer with tools. Other, more broadly scoped questions, like *what code does a similar operation?,* has no tool support, but developers were good at answering them with search tools.

Developers answered questions about causality ($u3$) such as *where did this value come from?* {ATU} and *how did the program arrive at this method?* {AM}, by a series of lower level questions, such as *what thread is the program in right now?* {AT}, *what is the value of this variable or data structure now?* {AEMTU}. (Sillito, Murphy and De Volder report similar indirect questioning [1].) This was done primarily with breakpoint debuggers, which required developers to translate their questions into an awkward series of actions:

A: Here we're formatting WSTValue…I can't do highlighting, so I go to Source Insight. Find where I am in devns—this is the guy that screwed up. Shift F8, highlight all occurrences, where it gets its value from. Here's where we set it. So I want a breakpoint here.

As developers refined their hypotheses, they changed their concern from the behavior of the existing system, to the hypothetical behavior after some change:

T: There's no file there, so something forgot it and I have a suspicion of what it is. Might mean that the free code has to get moved later.

Intuition was essential in answering all of these questions. The cost of testing hypotheses and the risk of a false hypothesis often prevented developers from finding a root cause. Instead, developers frequently assessed the value in continuing their investigation, stopping when they were satisfied {ATU}.

5.6 Reasoning about Design

Developers sought four kinds of design knowledge:

(d_1) *What is the purpose of this code?*
(d_2) *What is the program supposed to do?*
(d_3) *Why was this code implemented this way?*
(d_4) *What are the implications of this change?*

The purpose of code (d_1) was often unclear when developers found an API to use: Is it a public artifact or intended only for a particular component? Is it regularly maintained or no longer used? Some developers inferred purpose by finding example API uses {GHK}; sometimes they directly asked the code's author {T}.

Developers needed to know what the program was supposed to do (d_2), for example, to evaluate the correctness of a variable's value {ABCDFMR}:

> D, yelling across the hall: Is 'B' not a legal license key letter?

Sometimes this assessment was obvious. For example, a crash in a basic use case must be unintended. In other cases, what a program was supposed to do was an explicit, documented decision:

> M: I just want to double check and make sure the convert key only shows up in languages that it's supposed to, based on the spec.

It was rarely sufficient to understand the cause of a program behavior. Developers also needed to know the historical reason for its current implementation (d_3) {AEHRTV}. For example, when assessing whether a variable's value was "wrong," developers had to consider whether the value was anticipated by the designer and explicitly ignored or whether it was overlooked. They would do this by investigating the code's change history {AT} or by looking for bug reports that contained hints about its current design {ET}. Developers would seek this design rationale from the author of the code through face-to-face conversation or some other means {TV}, but in one case the author was unavailable {T}. Even when developers found a person to ask, identifying the information that they sought was hard to express, as developers struggled to translate detailed and complex runtime scenarios into words and diagrams.

The consequences of decisions were also important (d_4). For example, when triaging, developers often discussed hypothetical scenarios {FHKLRV}:

> V's teammate: Let's go ahead and block and make it into a single operation.
>
> V: But the upgrade script needs to look for individualization.

Design knowledge of all types was scattered among design documents {M}, bug reports {AHV}, and personal notebooks {AHMT}. Email threads sometimes contained design rationale {CFJ}, but were not shared globally. Code comments sometimes contained design rationale {H}, but developers hesitated to write them because of the cost of submitting code changes. Developers rarely searched these sources, because such sources were thought to be inaccurate and out of date:

> H: Given that I'll be the one fixing the bugs, I need to make sure I know not what we are doing, but why we are doing it. We have these big long design meetings, and everybody states their ideas, and we come to a consensus, but what never gets written in the spec is why we decided on that. Keeping track of that is really hard.

These problems led all but two developers to defer decisions because of missing design knowledge.

5.7 Maintaining Awareness

Developers worked to keep track of hardware, people and information needed for their tasks:

(a_1) *How have resources I depend on changed?*
(a_2) *What have my coworkers been doing?*
(a_3) *What information was relevant to my task?*

Some awareness information was "pushed" to developers through IM clients and alert tools {BDEFLMN}, and through check-in emails {CFJ}. Developers obtained other types of awareness by actively seeking it. One group had brief meetings throughout the day, to keep aware of problems that teammates were working on and issues on which they were blocked {M}; other groups had weekly meetings to keep awareness about triage and design choices. Developers would stop by coworkers' offices to update them on problems or to see what problems they were facing {AFGHJKLMRT}:

> F: I talked to [Joe] a bit about the execution, and gather objects is on track, but I still need to make the base class.
>
> F's boss: Yeah, [Joe] talked to me about it. We need to make sure files are not delay assigned. He's in this big whoop-de-doo about it.

Developers tracked their time and others', checking their calendars, glancing at schedules and asking their managers about priorities {BCK}. Managers communicated to their developers about upcoming changes in informal meetings, email announcements, or planning meetings {FL}. Because developers were often interrupted, they also sought awareness about their own work (a_3):

> G: Sometimes I have like 20 windows, 5 or 6 build windows, each one is a state that I'm working on and I lose it! If I could just save it…I would be really happy! I hate those midnight reboots.

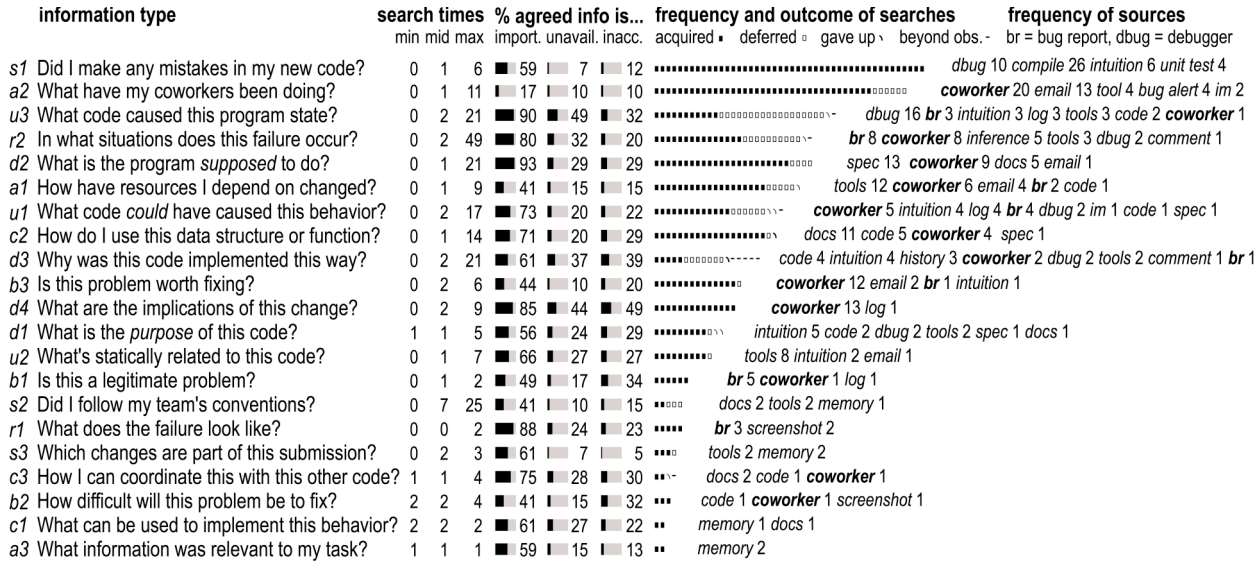

Figure 3. Types of information developers sought, with search times in minutes; perceptions of the information's importance, availability, and accuracy; frequencies and outcomes of searches; and sources, with the most common in boldface.

6. Quantifying Information Needs

The information needs we have discussed are summarized in Figure 3. The *time spent searching*, *search frequencies*, *search outcomes*, and *source frequencies* are based on our observational data. The outcomes include when developers *acquired* information, *deferred* a search with the intent of resuming it, or *gave up* with no intent of resuming it; a few searches continued beyond our observations. Also, in two cases, a need was initially deferred, then satisfied afterward by a coworker's email response; we coded these as *acquired*.

The most frequently sought and acquired information includes whether any mistakes (syntax or otherwise) were made in code and what a developers' coworkers have been doing. The most often deferred information was the cause of a particular program state and the situations in which a failure occurs. Developers rarely gave up searching. There was no relationship between deferring a search and whether the source involved people (bug reports, face-to-face, IM, email) ($\chi^2(1)=.6$, $p > .05$).

Based on medians, the information that took the longest to acquire was whether conventions were followed (*s*2); based on maximums, the longest to acquire was knowledge about design (*d*2, *d*3) and behavior (*u*1, *u*3). No one source of information took longer to acquire than another ($F(17, 321)=.53$, $p>.05$), nor was there a difference in search times between sources involving people and sources that did not ($F(1, 339)=.07$, $p>.05$). These times are misleading, however, as many of the maximums were on deferred searches, so they were likely longer than shown here. Further, developers gave up or deferred searches because they depended on a person known to be unavailable. They were also expert at assessing the likelihood of the search succeeding and would abandon a search if the information was not important enough.

6.1 Rating Information Needs

The percentages in the middle of come from a survey of 42 different developers (of 550 contacted), asking them to rate their agreement with statements about each of these information types, based on a 7-point scale from strongly disagree to strongly agree. The bars represent the percent of developers who agreed or strongly agreed that the information was (from left to right) *important to making progress*, *unavailable or difficult to obtain*, and *had questionable accuracy*.

The survey results reveal interesting trends. The majority of developers rated the most frequently sought information in our observations as more important, and they also rated frequently deferred information as more unavailable. One discrepancy is that developers rated coworker awareness (*a*2) as relatively unimportant, which conflicts with its frequency in our observations. It may be that coworker awareness is so frequent sought and successfully obtained that developers do not think about it. We also observed developers successfully obtain knowledge about the implications of a change (*d*4), whereas developers rated it relatively difficult to acquire. The survey also begins to reveal which information types have more questionable accuracy, namely knowledge about design (*d*2, *d*4), behavior (*u*1), and triage (*b*1, *b*2).

6.2 Most Common Information Needs

For each information need, Figure 4 lists those participants who had that need at least once during their observations. The most common need across participants was coworker awareness. Most of the information needs occurred among several developers from different teams in different business divisions, which suggests that these are representative of development work in general. A few of the information needs occurred for only a few participants, which suggests that this list is not complete. Observing more developers over longer periods of time could reveal other less frequent needs. (Had we not observed V, for instance, we would not have seen need *a*3.)

6.3 Unsatisfied Information Needs

Many of the frequent information needs are problematic, in that searches for the information were often satisfied (deferred or abandoned) and had long search times. The most frequently unsatisfied information needs were the following, with their percentage of unsatisfied queries and maximum observed search times:

1	*What code caused this program state?*	61%	21 min
2	*Why was the code implemented this way?*	44%	21 min
3	*In what situations does this failure occur?*	41%	49 min
4	*What code could have caused this behavior?*	36%	17 min
5	*How have the resources I depend on changed?*	24%	9 min
6	*What is the program supposed to do?*	15%	21 min
7	*What have my coworkers been doing?*	14%	11 min

This ranking may reflect that 11 of the 17 participants' teams were in a bug fixing phase. In particular, the information needs ranked 1, 3 and 4 are largely about bug reproduction and the ones ranked 2 and 6 are largely about evaluating possible fixes for bugs. Nonetheless, the fact that these information needs are so often unsatisfied and take such a long time clearly hindered developer productivity.

*a*2	*What have my coworkers been doing?*	15	ABCDEFHJKLMNRTU
*u*3	*What code caused this program state?*	11	ABDEFGMNRTU
*a*1	*How have resources I depend on changed?*	10	ACEFGJKLRT
*u*1	*What code could have caused this behavior?*	9	AEFGLMRTU
*c*3	*How do I use this data structure or function?*	9	ADFGHJKLN
*s*1	*Did I make any mistakes in my new code?*	9	ABDFHJKNR
*d*2	*What is the program supposed to do?*	7	ABCDFMR
*r*2	*In what situations does the failure occur?*	7	ELMRTUV
*b*3	*Is this problem worth fixing?*	7	BCJLRTV
*u*2	*What's statically related to this code?*	6	AEHKNT
*d*3	*Why was this code implemented this way?*	6	AEHRTV
*d*4	*What are the implications for this change?*	6	FHKLRV
*r*1	*What does the failure look like?*	5	AMNRT
*c*3	*How can I coordinate this with the other code?*	4	AHKN
*s*2	*Did I follow my team's conventions?*	4	BELN
*d*1	*What is the purpose of this code?*	4	HKT
*b*2	*Is this a legitimate problem?*	3	BLR
*s*3	*What changes are part of this submission?*	2	JN
*b*2	*How difficult will this problem be to fix?*	2	LR
*a*3	*What information was relevant to my task?*	1	V

Figure 4. Information needs per participant.

7. Discussion

Our motivation for this study was to identify and characterize software developers' information needs. While the list we have identified may not be complete, it has several implications.

7.1 Coworkers as Information Sources

Coworkers were the most frequent source of information, accessed at least once for 13 of the 21 information needs and in 83 of the 334 instances of information seeking. The importance of coworkers as information sources probably explains why coworker awareness is the second most frequent information need. (Developers checked on coworker availability almost as many times as they looked at output from the compiler or debugger.) This is consistent with prior work on awareness in software development, with regard to sources, strategies, and frequency of information seeking [5][9][15].

Why should developers turn so often to coworkers? One possibility is the topics being discussed. Outside of awareness, the information needs where coworkers were most often consulted were either about design, i.e.

- *What are the implications of this change?* (13 times)
- *What is the program supposed to do?* (9)
- *Why was the code implemented this way?* (2)

or about execution behavior, i.e.

- *Is this problem worth fixing?* (12)
- *In what situations does this failure occur?* (8)
- *What code could have caused this behavior?* (5)

In several instances coworkers were unavailable for these questions, and the developers' tasks were blocked once they sent their questions via email {ABCEFJR}.

Developers consulted coworkers about design because in most cases, design knowledge was only in coworkers' minds. The lack of design documentation may be due to inadequate notations, particularly for design intent and rationale. Two of our observed developers did have design documentation—a prototype for a user interface, a syntax grammar for a parser—which answered some of their questions {MG}. However, they still turned to coworkers when they questioned the accuracy of the documents.

Questions about program behavior were difficult to acquire because of the number of possible explanations. Developers had to use primitive tools to search this ex-

planation space, and so searches were driven by intuition or expert opinion. Developers also went to great lengths to learn behavior information. As one example, tofix a bug recently assigned to him, E had a tester nine times zones away reproduce the bug (at 2 am) since no one else had the right machine configuration. Because behavior information was hard to acquire, developers made triage decisions quickly based on implementation concerns and resource availability, rather than the organization's overall goals {BCJLRT}. That is, developers would favor those tasks with the fewest information needs.

7.2 Automating Information Sources

One approach to reducing this communication burden is to automate the acquisition of information. Given the frequent desire for awareness information, it is no surprise that researchers are already creating awareness displays for development teams, like FASTDash [1] and Palantír [17].

For example, many of developers' questions about static relationships depended on metadata such as build numbers and version histories, but developers manually incorporated such data in their searches. Similarly, tools for analyzing programs' dynamic behavior only partially helped with determining the cause of a program state; the rest had to be determined by hand using a breakpoint debugger and through guesswork. Task-specific applications of program slicing would be a way to automate some of this searching [19]. Implementation questions (*c*1, *c*2, *c*3) also lacked adequate tools (it is worth noting that these needs have also been discussed relative to end-user programming [10]). Tools were often appropriated for unanticipated uses, so it is within tool designers' interests to design tools that are amenable to appropriation. This might entail using standards, so that information may be passed between tools and transformed as needed.

Some information seeking cannot be automated because the information is currently unavailable. For example, when developers could not reproduce a failure, there was little they could do to find it. Tracing tools that can record the failure context would be a major advance. Failures could be debugged separately from their original context and the trace could be analyzed by multiple people. Design intent was also difficult to find. Information about rationale and intent existed sometimes in unsearchable places like whiteboards and personal notebooks or in unexpected places like bug reports. Some awareness information is difficult to acquire, for example, developers often wondered who is reading their code. Tools could make this available.

Aside from tools, one could address these information needs through process change, for example Agile methods. The frequent need to consult coworkers for information is an important motivation for Scrum meetings and radical collocation. Chong and Siino recently compared interruptions among radically collocated pair programmers versus cubicle-base solo programmers and found that the Agile team's interruptions were shorter, more on-topic, and less disruptive [4]. Our data about the importance of design knowledge provide evidence about the value of prototyping in software design, as well as the value of prototypes during implementation. Our observations about the importance of error checking, coupled with the distributed nature of design knowledge, also support the claims of pair programming: with two developers, with slightly different design knowledge, errors seem more likely to be caught or even prevented.

Last is the issue of notations for software design. While there is already considerable research on architecture description languages, UML, and various forms of model checking, our observations raise several pertinent questions. What can be written down cost-effectively? How can it be written to be searchable and so that its accuracy and trustworthiness are assessable by developers who consult it? It is worth noting that several participants perceived that face-to-face meetings to be a pleasant and efficient way to transfer design knowledge. The frequent conversations promote camaraderie and no effort is wasted recording design information that might never be read or might go stale before being read. Hence, a demand-driven approach to recording design knowledge might succeed over an eager "record everything" approach.

7.3 Study Limitations

Because we performed this study in the context of developers' real work, the external validity of our results are high. The diversity of our sample gives us confidence in generalizing across different products, team structures and development phases within the organization we studied. We were unable to control for corporate culture, although the communication patterns and development processes we observed are consistent with studies of other corporations. Other variations, such as testing practices, the talent and expertise of a company's developers, and more or less formal development processes, may have biased our findings.

Studies that rely on observations are subject to observers' biases, so it was essential that we have multiple observers. For example, we may have misunderstood what developers were looking for. To minimize intrusion, we chose to have a single observer per session, who took only written notes. (In several cases, the participants worked in shared or noisy offices.) Even with a single

observer, there were several instances of missed interruptions, where a visitor peeked inside the office, saw the observer, and chose to leave rather than interrupt. There were also many information seeking tasks that we could not observe because they were either too subtle, like glancing at a coworker's IM status, or invisible, like the use of memory to recall facts about the code. Our data was also biased by those issues that developers chose to mention during think aloud.

The timestamps in our logs are accurate within the minute, but are subject to typical clerical errors during transcription and copying. Also, some time was spent talking to the observers, but this bias was likely distributed throughout our observations. We only had a single coder categorize our logs, which affects our quantitative data; however, we feel the orders of magnitude in our data are accurate.

8. Conclusions

Our goals in this study were to identify software developers' information needs and characterize the role of these needs in developers' decision making. What we found were 21 types of information. Some of these were easy to satisfy accurately (awareness) but others with only questionable accuracy (the value of a fix and the implications of a change). Other needs were deferred often (knowledge about behavior and design), whereas some were impossible to satisfy in certain cases (reproduction steps). Not only do these needs call for innovations in tools, processes, and notations, but they also reveal how the collective responsibility for design knowledge can lead to intense awareness and communication needs observed in this study and others.

There are many future directions for this work. One issue we did not investigate were the decisions that developers made and the true accuracy of the information on which they were based. We have also proposed some explanations for the needs we observed, which should be tested. There are other populations, namely testers and architects, whose roles and information needs should also be studied. We hope that these investigations and others will bring us a more complete understanding of software development work and eventual improvements in software quality.

Acknowledgements

We extend our thanks to the developers who participated in our study for their valuable time. We also thank the VIBE, HIP and Visual Studio User Experience teams at Microsoft for their feedback. The first author was an intern at Microsoft Research over the summer of 2006.

References

[1] Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G., Bailey, B. (2007). FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. To appear at CHI 2007.

[2] Brooks, F.P. Jr. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, Reading, MA.

[3] Cataldo, M., P. Wagstrom, J.D. Herbsleb, K. Carley (2006). Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Computer Supported Cooperative Work 2006*, Banff, Alberta, 353–362.

[4] Chong, J., Rosanne Siino. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. *Computer Supported Cooperative Work 2006*, Banff, Alberta. p.28–39.

[5] de Souza, C.R.B., D.F. Redmiles, G. Mark, J. Penix, M. Sierhuis (2003) Management of Interdependencies in Collaborative Software Development: A Field Study. *ISESE*, Rome, Italy, 294–303.

[6] Eisenstadt, M. (1997). "My Hairiest Bug" War Stories. *CACM*, 40(4), 30–37.

[7] Gonzalez, V., G. Mark. , J. Harris (2005). No Task Left Behind? Examining the Nature of Fragmented Work. *CHI*, Portland, OR, 321–330.

[8] Gutwin, C., R. Penner, K. Schneider, K. (2004). Group Awareness in Distributed Software Development. *CSCW*, Chicago, IL, 72–81.

[9] Hertzum, M. (2002). The Importance of Trust in Software Engineers' Assessment of Choice of Information Sources. *Information and Organization*, 12(1), 1–18.

[10] Ko, A.J., B.A. Myers, H.H. Aung (2004). Six Learning Barriers in End-User Programming Systems. *VL/HCC*, Rome, Italy, 199–206.

[11] Ko, A.J., B.A. Myers, M.J. Coblenz, H.H. Aung (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *TSE*, 971–987.

[12] McDonald, D.W., M.S. Ackerman (1998). Just Talk to Me: A Field Study of Expertise Location. *CSCW*, Seattle, WA, 315–324.

[13] LaToza, T.D., G. Venolia, R. DeLine. (2006). Maintaining Mental Models: A Study of Developer Work Habits. *ICSE*, Shanghai, China, 492–501.

[14] Perlow, L.A. (1999). The Time Famine: Toward a Sociology of Work Time. *Administrative Science Quarterly*, 44(1), 57–81.

[15] Perry, D.E., N.A. Staudenmayer, L.G. Votta (1994). People, Organizations and Process Improvement. *IEEE Software*, July, 36–45.

[16] Sandusky, R.J., L. Gasser (2005). Negotiation and Coordination of Information and Activity in Distributed Software Problem Management. *GROUP*, Sanibel Island, FL, 187–196.

[17] Sarma, A., Z. Noroozi, A. van der Hoek, Palantír: Raising Awareness among Configuration Management Workspaces. *ICSE, 2003*, Portland, OR, 444–454.

[18] Seaman, C.B., V.R. Basili (1998). Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *TSE*. 24(7), 559–572.

[19] Sridharan, M., S.J. Fink, R. Bodik. Thin Slicing. To appear at *PLDI 2007*.

[20] Sillito, J., G. Murphy, K. De Volder (2006). Questions Programmers Ask During Software Evolution Tasks. *SIGSOFT/FSE*, Portland, OR, 23–34.