Information Needs in Collocated Software Development Teams

Andrew 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
{rob.deline, ginav}@microsoft.com

Abstract

Previous research has documented the fragmented nature of software development work, with frequent interruptions and coordination. To explain this in more detail, we analyzed software developers' day-to-day information needs. We observed seventeen developers at a large software development company and transcribed their activities minute by minute 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 identify twenty-one information types and catalog 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 sources of knowledge were 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 [1]. 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 [9], and bugs can arise from large chasms between the cause and the symptom, sometimes making debugging tools inapplicable [4].

One approach to understanding why these activities are so difficult is to understand them from an *information* perspective. For instance, some studies have investigated information *sources*, such as people [11], code repositories [3], and bug reports [14]. Others have studied means of *acquiring* information, such as e-mail, IM, and informal conversations [13]. Studies have even characterized developers' *strategies* [7], for example, how they decide who to ask for help.

While these studies have provided several concrete insights about features 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 find such information more quickly.

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 find this information?
- What inhibits the acquisition of such information?

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

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

2. Related Work

In past decades, there has been a great effort to understand social and organizational processes in software development. These studies have discovered a number of intriguing patterns, many of which reflect underlying information needs in developers' work.

For example, one undisputed feature of software development is its social nature. Perry et al. reported that over half of developers' time was spent interacting with coworkers [13]. Much of this communication is to maintain awareness. For example, de Souza et al. found that developers send e-mails before submitting code a check in to allow their peers to prepare for changes [3]. Collocation is a central factor in determining the quality of awareness information. For instance, Seaman et al. [15] found that local mobility facilitates awareness in ways that are unavailable in distributed situations. Similarly, coordination problems can be exaggerated across sites because of the lack of spontaneous communication channels [6].

Developers also communicate to obtain knowledge [7]. For example, LaToza et al. describe the role of the "team historian," who possesses knowledge about the origins of a project and its architecture [11]. To determine who to ask, developers often gauge expertise by inspecting check in logs [3], but such information is not always accurate [10].

One consequence of developers' information needs is the fragmentation of time. Gonzalez et al. found that developers average about 3 minutes on a task and about 12 minutes in an area of work before switching [5]. These switches occur due to reassignment to a higher priority task and getting blocked [13]. Perlow found that a software group's collective use of time perpetuated a "time famine" with the feeling of having too much to do and not enough time to do it, largely due to needless interruption [12].

Dependencies are also a central factor in software development work. For example, developers use bug reports, content management systems, and version control systems to manage dependencies and notify coworkers of new dependencies [3]. Teams will "fork" software versions to avoid a team dependency, even though they later have to duplicate fixes to the forked code [11]. Developers will also rush their activities to minimize dependencies between their code and recently committed changes in the repository [3].

Although all of these factors are important in determining the nature of software development work, we know little about *why* they are important. By studying developers' information needs, we can begin to explain the patterns observed in prior work.

3. Method

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

Each observation session was roughly 90 minutes and, to minimize invasiveness, used only one observer. At the beginning of each observation session, 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 lower level events like keystrokes or menu selections. Since we shared the participants programming background, we could understand 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.

After meeting with 17 developers, both observers felt they had seen enough redundancy that additional sessions were unnecessary. Details about the developers we observed are shown in the left of Figure 2, which introduces initials that will be used to refer anonymously to specific developers throughout this paper. *Dev leads* manage *SDEs* (s*oftware development engineers)* while also performing a development role. Our sample is random in several respects, including experience, phase, and customer (including projects with no dependent teams, projects on which many teams depended, and several in between).

4. Task Structure

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

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 of J's observation log.

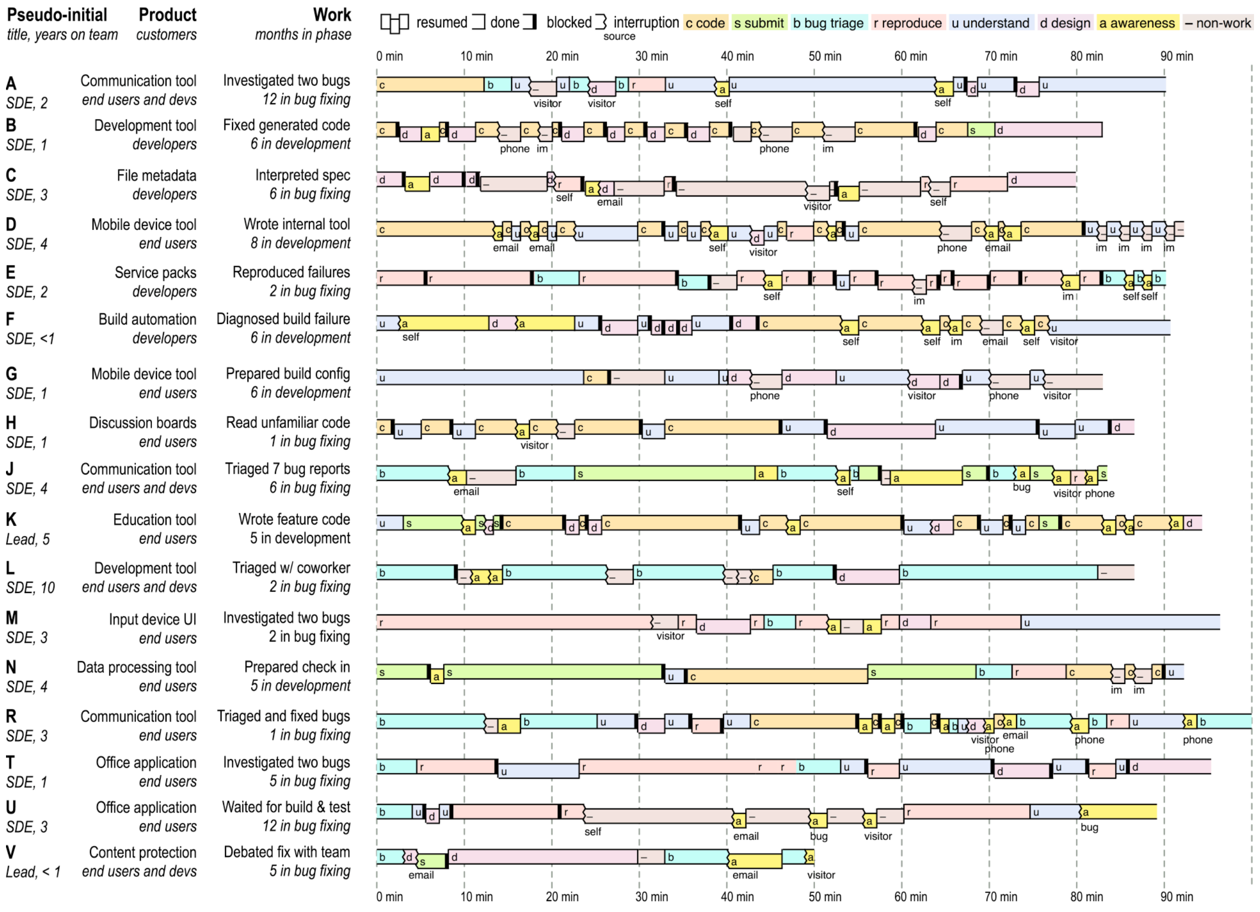


Figure 2. The backgrounds and task structures of the 17 observed developers. Tasks are labeled time blocks with coded ends (thin line for *done*, thick line for *blocked*, jagged line for *interrupted*). When a task gets broken up by interruptions/blocking, we draw its fragments at the same level to show resumption.

based on remarks like "I want to get back to my repro…" All but the last two of these causes of task switches are interruptions. 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 as *task avoidance* if she could but chose to switch anyway.

Figure 2 visualizes these switches, which occurred an average of every 5 minutes (± 1.7) (mirroring the rate reported in Gonzales and Mark [5]). Time fragmentation varied considerably per participant. For example, M reproduced failures uninterrupted, whereas R was frequently blocked. Much of this was due to conversations, many of which were face to face or over IM or phone. Developers had between 0 and 6 conversations per session (median of 1). Developers were also interrupted by notifications, such as e-mail and bug change alerts (which a tool presented when a bug report changed):

> CMS doesn't like that 'cause it doesn't … Ah! Someone just assigned me a bug…Oh good, this is the bug that I just submitted to triage. {U}

Developers experienced between 0 and 9 notifications per session (median of 1).

Blocking, shown in Figure 2 as dark vertical bars, occurred when some information was unavailable. Some resources were computationally bound, such as compilation and tests suites. Developers also waited for e-mail replies about opinions or schedules and for other teams to submit changes or fix bugs. Other blocks were due to missing knowledge, such as when a developer would have to stop coding to learn about an API. Developers were blocked a median of 4 times per session and between 0 and 11 times overall.

Since a developer can retain only so much information, blocking on information needs and interrupting tasks for new information are unavoidable and not inherently unproductive. Similarly, frequent task switching can have many causes (including personal work style) and is not inherently unproductive. Given the variety of the participants and tasks we observed, we did not attempt to measure their productivity or otherwise judge their task structure.

5. Information Needs

Next we describe the *information needs* of each work category. We list the initial of the developers for whom we observed a trend within braces.

5.1 Writing Code

Writing code often required knowledge beyond that which developers already possessed:

c1. What data structures or functions can be used to implement this behavior?
c2. How do I use this data structure of function?
c3. How can I coordinate this 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 *static* usage rules (*c2*). For example, which method is appropriate to call? What data structures does this require? What constructors does this class have? In cases where documentation was available, developers sought it {DFHJKN}, but there were situations where they needed to use code that was only fully understood by the author {AFL}. Others found uses of the code from which to infer these rules {GH}.

Because developers had to coordinate APIs with their own code, they also sought *dynamic* usage rules (*c3*), 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 documented. Developers used their colleagues {A}, documentation {K}, and also example code {HN} to infer these rules.

5.2 Submitting a Change

Whether bug fixing or feature writing, developers had three primary questions when submitting code:

s1. Did I make any mistakes in my new code?
s2. Did I follow my team's conventions?
s3. Which changes are part of this submission?

Besides building their code to assess its *syntactic* correctness, developers answered questions of correctness (*s1*) by considering the scenarios and range of input that they had 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:

> 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. {K}

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

> Okay, here we go...Gah! I never want to see word asserts but they always pop up. They have nothing to do with my work! {A}

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 (*s2*). Some teams required tags or other documentation to be embedded in method headers and developers were careful to remember to insert these, often with the help of tools {BELN}. Sometimes two submissions intersected (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 (*s3*) {JN}. This was a manual process, in which developers tried to remember which change task they belonged to.

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:

b1. Is this a legitimate problem?
b2. How difficult will this problem be to fix?
b3. Is it worth fixing?

Assessing legitimacy (*b1*) involved deciding whether a failure was due to a problem with the code or an unrealistic configuration of a test {BL}:

> 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. {B}

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}:

> 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...{A}

When developers assessed a report's difficulty (*b2*), they 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 would often fix bugs "by design," treating them as work items for later releases, as in this dialogue between two developers:

> I think the best thing is a new overlay to indicate something's going on.
>
> We can't do that by the release. Looks like a work item {V}.

Another factor affecting difficulty 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}.

When assessing whether a failure was worth fixing (*b3*), developers considered the number of users would be affected {CJLV} and the user experience {LRV}. For example, Victor's discussed a fix:

> If we want to push it back we can, but I think an overlay is the easiest to do {V}.
>
> But it's a totally broken experience for the user.

If there was a known workaround developers often focused on more severe bugs {LV}.

5.4 Reproducing the 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 the bug report. 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}:

> 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 ... {A}

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:

> 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. I'm going to try group by. {A}

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 who might attempt to reproduce the failure in the future.

5.5 Understanding Behavior

Aside from understanding their own code, developers had to understand unfamiliar code when using vendor code {GM}, joining a new team {V}, obtaining ownership of code {H}, balancing workload {T} and during debugging, when unfamiliar code was encountered 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:

> 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. {A}

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 *static* properties of the structure of the code (*u2*), such as what is the definition of this and what calls this method? Such questions were easy to answer with tools. Other more broadly scoped static questions, such as *what code does a similar operation?* were unsupported by tools, but developers were good at answering them with text 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 answering 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}. This was done primarily with breakpoint debuggers, which required developers to translate their questions into an awkward series of actions:

> Here we're formatting WSTValue...I can't do highlighting, so I go to Source Insight. Find where I am in dev ns—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. {A}

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

> 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 {T}

In answering all of these questions, intuition was essential. The cost of testing hypotheses, and the risk of a hypothesis being false, meant that developers did not often have time to find the root cause. Instead, developers frequently assessed the value in continuing their investigation, stopping when necessary {ATU}.

5.6 Reasoning about Design

Developers sought four kinds of design knowledge:

d1. What is the purpose of this code?
d2. What is the program is supposed to do?
d3. Why was this code implemented this way?
d4. What are the implications of this change?

The purpose of code (*d1*) was often unclear when developers found part of an API to use: was it a public artifact, or only to be used by a particular component? Is it being regularly maintained or is it no longer used? Some developers inferred purpose by examining example uses of the API {GHK}, but sometimes they asked the authors directly {T}.

Developers needed to know what the program was *supposed* to do (*d2*), for example, to evaluate the correctness of a variable's value {ABCDFMR}:

> [from across the hall] Is 'B' not a legal license key letter? {D}

Making this assessment was often obvious, for example, developers knew immediately that a crash in a basic use case was unintended. In other cases, what a program was supposed to do was an explicit, documented decision:

> 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 enough to understand the cause of a program behavior. Developers also needed to know the historical reason for its current implementation (*d3*) {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 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 they were unavailable {T}. Even when developers found a person to ask, identifying the information that they sought with words was often difficult, as developers struggled to translate detailed and complex runtime scenarios into words and diagrams.

The *consequences* of decisions were also important (*d4*). For example, when triaging, developers often discussed hypothetical scenarios {FHKLRV}:

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

Design knowledge of all types was scattered among design documents {M}, bug reports {AHV}, and personal notebooks {AHMT}. E-mail threads contained the reasons for design choices {CFJ}, but they were not shared globally. Comments often contained design rationale {H}, but developers were hesitant to write them because of the cost of submitting changes. Developers rarely searched these sources, because such sources were thought to be inaccurate and out of date. Hank described this problem well:

> 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. Keepin' track of that it's really hard to do. {H}

These problems led all but two developers to defer a decision because of a lack of design knowledge.

5.7 Maintaining Awareness

Developers sought three types of awareness:

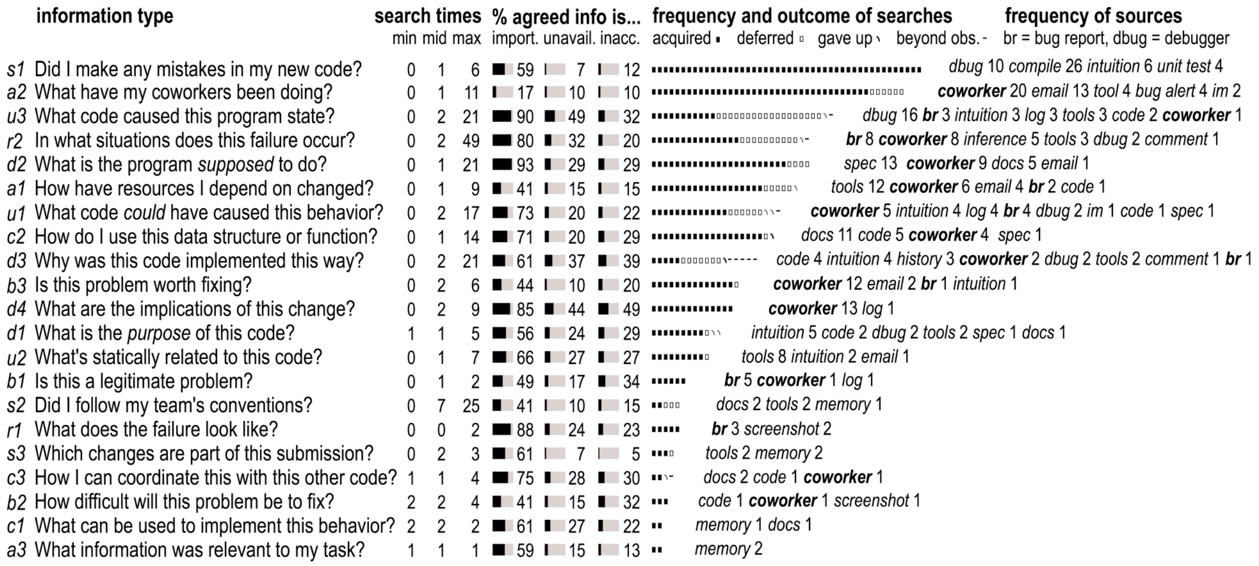
a1. How have resources I depend on changed?
a2. What have my coworkers been doing?
a3. What information was relevant to my task?

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

> I talked to Jason a bit about the execution, and gather objects is on track, but I still need to make the base class.
>
> > Yeah, Jason talked to me about it. We need to make sure files are not delay assigned. He's in this big whoop dee doo about it {F}

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, e-mail announcements, or planning meetings {FL}. Because developers were often interrupted, they also sought awareness about their own work (*a3*):

> 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. {G}



	information type	search times min	mid	max	% agreed info is... import.	unavail.	inacc.	frequency of sources (br = bug report, dbug = debugger)
s1	Did I make any mistakes in my new code?	0	1	6	59	7	12	*dbug* 10 *compile* 26 *intuition* 6 *unit test* 4
a2	What have my coworkers been doing?	0	1	11	17	10	10	***coworker*** 20 *email* 13 *tool* 4 *bug alert* 4 *im* 2
u3	What code caused this program state?	0	2	21	90	49	32	*dbug* 16 ***br*** 3 *intuition* 3 *log* 3 *tools* 3 *code* 2 ***coworker*** 1
r2	In what situations does this failure occur?	0	2	49	80	32	20	***br*** 8 ***coworker*** 8 *inference* 5 *tools* 3 *dbug* 2 *comment* 1
d2	What is the program *supposed* to do?	0	1	21	93	29	29	*spec* 13 ***coworker*** 9 *docs* 5 *email* 1
a1	How have resources I depend on changed?	0	1	9	41	15	15	*tools* 12 ***coworker*** 6 *email* 4 ***br*** 2 *code* 1
u1	What code *could* have caused this behavior?	0	2	17	73	20	22	***coworker*** 5 *intuition* 4 *log* 4 ***br*** 4 *dbug* 2 *im* 1 *code* 1 *spec* 1
c2	How do I use this data structure or function?	0	1	14	71	20	29	*docs* 11 *code* 5 ***coworker*** 4 *spec* 1
d3	Why was this code implemented this way?	0	2	21	61	37	39	*code* 4 *intuition* 4 *history* 3 ***coworker*** 2 *dbug* 2 *tools* 2 *comment* 1 ***br*** 1
b3	Is this problem worth fixing?	0	2	6	44	10	20	***coworker*** 12 *email* 2 ***br*** 1 *intuition* 1
d4	What are the implications of this change?	0	2	9	85	44	49	***coworker*** 13 *log* 1
d1	What is the *purpose* of this code?	1	1	5	56	24	29	*intuition* 5 *code* 2 *dbug* 2 *tools* 2 *spec* 1 *docs* 1
u2	What's statically related to this code?	0	1	7	66	27	27	*tools* 8 *intuition* 2 *email* 1
b1	Is this a legitimate problem?	0	1	2	49	17	34	***br*** 5 ***coworker*** 1 *log* 1
s2	Did I follow my team's conventions?	0	7	25	41	10	15	*docs* 2 *tools* 2 *memory* 1
r1	What does the failure look like?	0	0	2	88	24	23	***br*** 3 *screenshot* 2
s3	Which changes are part of this submission?	0	2	3	61	7	5	*tools* 2 *memory* 2
c3	How I can coordinate this with this other code?	1	1	4	75	28	30	*docs* 2 *code* 1 ***coworker*** 1
b2	How difficult will this problem be to fix?	2	2	4	41	15	32	*code* 1 ***coworker*** 1 *screenshot* 1
c1	What can be used to implement this behavior?	2	2	2	61	27	22	*memory* 1 *docs* 1
a3	What information was relevant to my task?	1	1	1	59	15	13	*memory* 2

frequency and outcome of searches
acquired ▪ deferred ▫ gave up ˎ beyond obs. -

Figure 3. Types of information developers sought, with search times, perceptions importance availability, and accuracy, frequencies and outcomes of searches, and sources. The most common sources are in bold.

6. Quantifying Information Needs

The information needs we have discussed are summarized in Figure 3, which includes data from two sources. 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; some searches continued beyond our observations. Also, in two cases, a need was initially deferred, then satisfied afterward by a coworker's e-mail response; we coded these as *acquired*.

The percentages in the middle of Figure 3 come from a survey of 42 developers (of 550 surveyed), 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 "important to making progress," "unavailable or difficult to obtain," and "had questionable accuracy."

Figure 3 reveals many interesting trends. The most frequently *sought* and *acquired* information include 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 searches for information. There was no relationship between whether the source involved people (bug reports, face to face, IM, and e-mail versus others) and whether the search was deferred ($\chi^2(1)=.6$, $p > .05$).

Based on medians, the information that took the longest to acquire was whether conventions were followed (*s2*). Based on maximums, the information that took longest to acquire was knowledge about design (*d2*, *d3*) and behavior (*u1*, *u3*). Based on this data, 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. Furthermore, developers gave up or deferred searches because they depended on a person that they knew was unavailable; they were also expert at assessing the likelihood of the search *succeeding* and would abandon a search if information was not important enough.

The survey results reveal other 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 interesting discrepancy is developers rated coworker awareness (*a2*) as relatively unimportant, which conflicts with our observations. It may be that coworker awareness is so fundamental to developers' work that they do not even consider it information. We also observed developers successfully obtain knowledge about the implications of a change (*d4*), 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 (*d2*, *d4*) behavior (*u1*), and triage (*b1*, *b2*).

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 Importance and Availability

Among the 21 needs we observed, there were many that appeared particularly important or challenging to satisfy. For example, *awareness* was one of the most frequently sought and found types of information (Figure 3). This is consistent with prior work on awareness is software development, with regard to sources, strategies, and frequency of information seeking [3][7][13].

Questions about the *design* of a program were difficult to acquire because in most cases, design knowledge was distributed among many developers and across many teams. This might explain why awareness was so important to developers' work: design knowledge had to be acquired from certain people, and thus developers needed to track these people's presence and decisions. Questions about the *behavior* of a program (how the program works from a logical viewpoint) were difficult to acquire because of the number of explanations for program's behavior. Developers had to use primitive tools to search this explanation space, and so searches were driven by intuition. Because information about behavior and design typically required communication, developers were often blocked once they sent their question via e-mail {ABCEFJLR}. When the design *was* documented, developers simply consulted the design and resumed their task, except when they did not trust its authority or thought it out of date {BM}.

Although most of the other types of information were easy to acquire, the accuracy of some was questionable. Most questions about APIs were answered by consulting documentation or coworkers, but such knowledge seemed suboptimal. Developers had efficient processes for error checking their code, but it was unclear whether this ever identified errors. Developers made triage decisions quickly, but based on implementation concerns and resource availability, rather than the organization's overall goals {BCJLRT}.

Some information was impossible to acquire. If a bug report did not have a clear description of what the failure looked like (*r1*), nor what situations it occurred in (*r2*), there was no other source for developers to acquire it from. When a developer needed to know the rationale for some code's design (*d3*) and the author was unavailable, the developer could only guess.

7.2 Explaining the Trends

Why was awareness so frequently sought in our observations? Why was design knowledge so difficult to acquire? What explains the frequency of fragmentation and blocking in developers' work in our study and in others [5]?

It is possible that these trends are unique to the organization we studied and the developers we observed. A more likely explanation, however, is that these trends are simply a feature of software development work that emerges from the inherent *complexity* in software artifacts. One consequence of this complexity, which seems not to occur in other engineering disciplines, is that there are few *notations* that can capture and document design knowledge (whether requirements, specifications, implementation, or architecture). This was evident from our observations: code did not look like design; intent could rarely be inferred from code; programming languages only allowed a single, structural perspective on code, yet there were many other perspectives on which developers reasoned about code. Because of the lack of notations for all of this knowledge that developers relied on, the knowledge was primarily stored in the minds of developers. Consequently, developers relied on each other for design knowledge, resulting in the blocking evident in Figure 2.

This explanation of the work we observed is consistent with other observations. For example, this might explain why small teams are thought to be more productive [1]: the smaller the team, the easier it is to maintain awareness, and therefore the easier it is to acquire knowledge from coworkers. Of course, conventions and the shared (or at least overlapping) understanding of code also meant that developers could communicate their points to team members efficiently. This might also explain the attitudes of several of the dev leads that we spoke to about the importance of *alignment* between technical dependencies in the code with social relationships between teams (also reported elsewhere [2]):

> Managers established communications, encouraged personal relationships, and engagement with [other teams]. The toughest issue is not with fixing a bug, it's when you depend on someone. You don't know if it's your problem or not. You're blocked. You can't plan ahead. All of the issues with dependencies are much tougher. {V}

This distribution of design knowledge among developers meant that individuals collaborated with different visions of the end result, and perhaps without a consistent vision. Because design was everyone's responsibility, developers had to keep the design in their minds synchronized through discussion. Of course, one benefit of this distribution of knowledge was that design was more malleable.

Two of our observed developers did have notations that covered important aspects of their product's designs {MG}. In these cases, design documentation—a prototype for a user interface, a syntax grammar for a parser—acted as oracles. Developers acquired design knowledge individually, like consulting a blueprint for a building, independent from their coworkers; as a result, they were not blocked. The tradeoff was that these documents had to be kept up to date and were more difficult to change. They also suffered from issues of trust: when developers consulted these documents, they sometimes needed to validate the accuracy of the information they acquired before proceeding {MG}.

7.3 What to Do?

Many of the information needs we observed were satisfied by consulting coworkers (Figure 3). This is not inherently bad: some types of information, such as awareness, might be best satisfied with personal interaction. However, that so many of the needs required communication is a potential source of productivity loss.

One approach to reducing this communication burden is to *automate* the acquisition of information. 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. Dynamic slicing tools would be a way to automate some of this searching. Implementation questions (*c1*, *c2*, *c3*) also lacked adequate tools (it is worth noting that these needs have also been discussed relative to end-user programming [8]). 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 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.

Some information is best made available through changes to *process*. For example, our observations about the importance of awareness information may explain the interest in agile methods, scrum meetings, and radical collocation that we observed {F}. 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.

7.4 Limitations

Because this study was performed in the context of developers' 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 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 and the talent and expertise of a company's developers, may have biased our findings.

Studies that rely on observations are subject to observers' biases, so it was essential that we have two independent observers. For example, we may have misunderstood what developers were looking for. There were also many information seeking tasks that we did not observe; in some cases needs were trivial to satisfy, such as glancing at a coworker's IM status; in others, the seeking was unobservable, such as a developer using their memory to recall some fact about code. Our data was also biased by issues that developers thought aloud about. We found that even note taking was too intrusive, preventing us from capturing higher fidelity (and thus more reliable) data. It also would have been too invasive to have multiple observers watching one developer because of the space limitations in offices, some of which were shared.

The timestamps in our logs are accurate within the minute but there may have been data entry errors that affected our time calculations. Furthermore, 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 likely 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 twenty-one 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.

9. Acknowledgements

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

10. References

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

[2] Cataldo, M., Wagstrom, P., Herbsleb, J.D., Carley, K. (2006) Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. CSCW, Banff, Alberta, to appear.

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] Wu, J., Graham, T. C. N., Smith, P. W. (2003). A Study of Collaboration in Software Design. *ISESE*, Rome, Italy, 304-315.