Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost

David Piorkowski[1], Austin Z. Henley[2], Tahmid Nabi[1],
Scott D. Fleming[2], Christopher Scaffidi[1], Margaret Burnett[1]

[1]Oregon State University
Corvallis, OR, USA
{piorkoda,nabim,cscaffid,burnett}@eecs.oregonstate.edu

[2]University of Memphis
Memphis, TN, USA
{azhenley,Scott.Fleming}@memphis.edu

ABSTRACT

Empirical studies have revealed that software developers spend 35%–50% of their time navigating through source code during development activities, yet fundamental questions remain: Are these percentages too high, or simply inherent in the nature of software development? Are there factors that somehow determine a lower bound on how effectively developers can navigate a given information space? Answering questions like these requires a theory that captures the core of developers' navigation decisions. Therefore, we use the central proposition of Information Foraging Theory to investigate developers' ability to *predict the value and cost* of their navigation decisions. Our results showed that over 50% of developers' navigation choices produced less value than they had predicted and nearly 40% cost more than they had predicted. We used those results to guide a literature analysis, to investigate the extent to which these challenges are met by current research efforts, revealing a new area of inquiry with a rich and crosscutting set of research challenges and open problems.

CCS Concepts

• Software and its engineering→Software notations and tools • Software and its engineering→Software creation and management

Keywords

Information foraging theory; navigation value and costs

1. INTRODUCTION

In a landmark paper in 2006, Ko et al. quantified the high cost developers incur when foraging for the information they need. In their results, developers spent 35% of their time on the *mechanics alone* of foraging between code fragments [21]. Expanding upon these findings, we showed that when other aspects of foraging were also taken into account, developers spent an average of 50% of their time foraging [32].

These and other findings about developers' foraging and navigations (e.g., [26, 31, 37, 44]) have led software engineering researchers to produce tools that help to reduce developers' navigation costs as they look for the information they need [8, 17, 18, 19, 22, 30, 33, 43]. Despite these gains, however, little is known at the foundational level of developer navigation—how well developers go about *choosing* where to navigate.

FSE'16, November 13–18, 2016, Seattle, WA, USA

http://dx.doi.org/10.1145/2950290.2950302

This is where theory can help. The essence of theories is abstraction—mapping instances of successful approaches to crosscutting principles. In the realm of human behavior, these abstractions can then produce explanations of *why* some software engineering tools succeed at supporting the efforts of software developers and why some tools that were expected to succeed did not.

As Shaw eloquently explained, scientific theory lets technological development pass limits previously imposed by relying on intuition and experience [40]. For example, her summary of civil engineering history points out that structures (buildings, bridges, tunnels, canals) had been built for centuries—*but only by master craftsmen*. Not until scientists developed theories of statics and strength of materials could the composition of forces and bending be tamed. These theories made possible civil engineering accomplishments that were simply not possible before, such as the routine design of skyscrapers by ordinary engineers and architects [40]. And indeed, in computer science, we have seen the same phenomenon. For example, expert developers once built compilers using only their intuitions and experiences, but the advent of formal language theory brought tasks like parser and compiler writing to the level that undergraduate computer science students now routinely build them in their coursework [1].

In this paper, we use Information Foraging Theory (IFT) as our theoretical foundation [34]. IFT provides a conceptual framework describing how people in an information environment, such as an IDE, seek information. For example, for developers faced with a bug, the information they seek may include how to reproduce the bug, what causes the bug, where to fix the bug, and whether similar bugs were fixed elsewhere [29].

According to IFT, developers make their navigation decisions by predicting the value a navigation will bring and the cost they will incur if they take that navigation. When they carry out these navigation decisions, they may be in for disappointments if (1) their destination does not provide as much value as expected or (2) the cost of extracting the information or getting to it is higher than expected.

This suggests that the fundamental issue behind navigations is how accurate developers are about predicting value and cost, and whether their accuracy is "enough" for them to be productive. To investigate this issue, we conducted an empirical study and literature analysis, grounded in IFT and structured according to the following research questions:

- RQ1 (*Value*): How often do developers' foraging decisions yield less *value* than they expect, and why?
- RQ2 (*Cost*): How often is the *cost* to gather and process desired information more than foraging developers expect, and why?
- RQ3 (*Trends in aligning actual value/cost with developers' expectations*): What aspects of the above questions do current SE research trends address, and how?

2. BACKGROUND AND RELATED WORK

2.1 Information Foraging Theory

In essence, Information Foraging Theory [34] gives a meaningful way to "carve up" an information environment for software development and the developers' actions within that environment, and then to investigate how developers use the environment. IFT has already proven useful for explaining and predicting developer behaviors during software maintenance, in ways beneficial to tool design [12, 16, 17, 25, 26, 27, 33].

IFT's main constructs are a *predator* (here, a developer) who seeks *prey* (the developer's information goals) within an information environment made up of information *patches* (in this paper, Java methods) connected by *links* (here, IDE features for navigating between methods; e.g., search-results links, clickable links to a method, adjacent methods via scrolling). Each link has a *cost* (time to get from one patch to the other) that is influenced by both system performance and the human's cognitive and physical speed. Fig. 1 shows the information environment in this paper.

Within each patch are *information features* (here, words, phrases, and graphics in the code or documentation), some of which may be the *prey* that the predator seeks. Information features have *value*, and they also have *cost* (e.g., time for the human to read and process them). Some of the information features are cues that label outgoing links to other patches. *Cues* (the labels on the links) provide the predator with hints about what information features may be found at the other end of the link. Fig. 2 conceptually illustrates two patches with information features, cues, and links. In modern development environments, like Eclipse, most displayed text has some form of clickable link (e.g., the Open Declaration shortcut on identifiers in the Editor), so there tends to be a high density of cues in such environments.

IFT's constructs are tied together via IFT's central proposition, which says that the predator treats foraging as an optimization problem. More specifically, according to IFT, the predator's foraging actions try to maximize the value V of information they will gain from a patch per their cost C of getting to and interacting with that patch, i.e.:

$$\max(V/C)$$

However, predators do not have perfect information about a patch's value and cost, so they make their choices based on their *expectations* of value V and of cost C, i.e.:

$$\max(E(V)/E(C))$$

The predator bases such expectations on whatever information they have gleaned so far, such as by inferring them from available cues.

How accurate are foragers in forming these expectations? Software engineering research has much to say about information professional developers seek (e.g., [26, 42]), but has not systematically considered the question of how well developers can *predict* these values and costs, or the accompanying implications for SE tools. These are the questions this paper investigates.

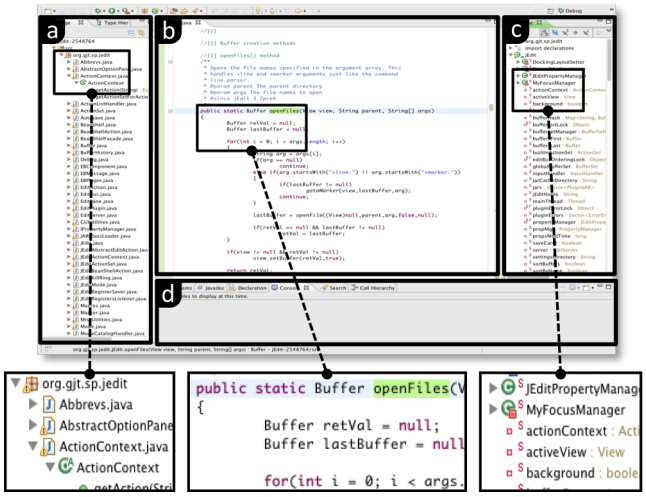

Fig. 1. An information environment (Eclipse) as a developer (predator) might see it during debugging. Patches of information content are visible in the (a) Package Explorer, (b) Editor, and (c) Outline View (plus a region (d) where other patches can appear). As the blow-up of (a) shows, each item in the Package Explorer is an information feature and also a cue, because the item has a link: clicking it opens a file. In (b)'s blow-up, the text "openFiles" in the Editor is also an information feature and a cue, because it has a clickable link.

2.2 Prior Empirical Studies of Developers

Prior empirical studies have observed the navigation behavior of developers during maintenance activities. That work has shed light on the different kinds of information that developers seek. These include the need for information about how to use APIs [10], about data or control flow [23, 46], and about requirements [31]. These studies have also revealed that developers seek this information to answer specific questions that they have [23, 42], to test mental hypotheses about how an existing program works [4], to develop an overall sense of context [13], or to follow leads that simply look relevant to their current task [26]. Rather than focusing on what developers seek and why, our investigation examines the extent to which they accurately *predict* the value of what they find in particular patches and the cost of getting it.

Often the reason for seeking code is to edit it, and several prior studies have investigated how developers edit code. For example, Ying and Robillard investigated whether developers make edits differently depending on whether they are fixing bugs or adding enhancements [47]. As another example, Posnett et al. investigated the extent to which developers make focused patterns of code edits across maintenance tasks (sometimes called "ownership" of code), and whether these patterns are statistically related to the resulting rates of defects [35]. In our study, although participants were editing at times, our research questions centered on foraging rather than on editing per se.

Little empirical work has investigated the specific question of developers' abilities to predict the value and cost of their navigations. One study examined navigations of analysts through

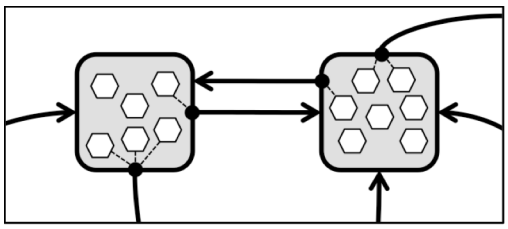

Fig. 2. Conceptual depiction of an information environment with two information patches (rounded boxes) interconnected by links (directed edges). Each patch contains information features (hexagons), with some that are also cues (connected to outgoing links).

documents and code as they attempted to recover requirement traceability between use cases and Java classes, revealing that the subjects spent a disproportionate amount of time in low-value patches (i.e., more time than the value of those patches justified) [31]. This result was consistent with an earlier study into the challenges of information foraging by developers, such as the fact that searches in code frequently failed to turn up desired results, and that developers spent substantial time organizing and reorganizing files and bookmarks in the IDE [21]. Another study of how developers search online resources corroborated that searches often lead to irrelevant code and supporting documents, except after developers had determined the right search terms for specific subtopics [2]. Our work builds on these results.

3. EMPIRICAL STUDY METHODOLOGY

To answer our first two research questions, RQ1 and RQ2, we conducted a think-aloud study of professional developers. The developers worked on a debugging task, and we recorded their work. We then collected developers' insights by playing back the recording for the developer and pausing it to ask them questions about key events. Through this method, we gathered data on the developers' navigation decisions and their assessment of expected vs. actual values and costs relative to those decisions, as they worked on the debugging task.

3.1 Participants, Procedures, and Task

Ten professional developers at Oracle participated in the study. The participants had 4.5–40 years of professional software development experience and 2–19 years of professional experience programming with Java specifically. We conducted the study one-on-one with each participant.

The sessions lasted no more than 2 hours. At the beginning of the session, participants filled out a background questionnaire. They then worked for 20 minutes on the debugging task. The task, by design, was sufficiently complex that no participant finished it in the allotted time. During the debugging task, we prompted participants to "talk aloud" as they worked so as to gather data on their information goals and intentions of their navigations. We recorded participants as they worked, capturing the computer screen, the participants' facial expressions, and their verbalizations.

The participants' task was to debug an actual bug (issue #3223) in the jEdit open source project. jEdit is a code editor written in Java, with code base of 98,652 non-comment lines. The bug was a problem with deleting "folded" text in the editor. Participants used the Eclipse IDE on a Windows PC to complete the task. We also allowed them to use any other tools they wanted to complete the task as they saw fit, including using the web.

After the debugging session, we conducted a retrospective semi-structured interview. The purpose was to collect participants' expected value before a navigation, and then the value they actually received. To collect the data, we played the session video for the participant, pausing it at each *to-method* event and *away-from-method* event on the video, to ask the questions shown in Fig. 3. The to-method pauses occurred just as the participant navigated to a Java method (just before arriving there), and the away-from-method pauses occurred just as the participant navigated away from the method (just before seeing the new location). If participants visited other kinds of files (e.g., properties files), we asked them the same questions as for the methods.

3.2 Qualitative Analysis

To analyze the data, we used a qualitative coding approach to map key concepts ("codes") to participants' navigations [39]. Specifically, we coded the videos (which included both navigation actions and corresponding verbalizations by the participants) whenever participants talked about the value or cost of a navigation.

For the purposes of this paper, we defined a *navigation* to a method to be any occurrence of the Eclipse editor's text cursor automatically moving to a method, or the participant scrolling to bring a method into view while also talking about the method. The *destination point* of a navigation was a method in the editor. The *starting point* of a navigation was any view in Eclipse from which a participant scrolled or clicked to arrive at a method. For example, selecting a search result or a link in an exception stack trace would open the corresponding code file in the editor, and place the text cursor within the relevant method.

We coded each navigation for which participants assessed value or costs as follows. First, two researchers iteratively refined the code set. Then, using the resulting code set on fresh data, they independently coded 20% of the data. Their resulting inter-rater reliability was 86% agreement using the Jaccard index (the intersection of all applied codes over the union of all applied codes) on 20% of the data for the value codes, and 81% agreement on 20% of the data for the cost codes. Given that rate of agreement, the coders then divided up the coding of the remaining data. We detail each code set in the Results sections that refer to them.

4. EMPIRICAL RESULTS

4.1 RQ1: Developers' Expectations of Value

4.1.1 Did They Get the Value They Expected?

To investigate the participants' assessments of a patch's value, we used an ordinal scale of measurement. That is, rather than attempting to quantify their value assessments numerically, we derived from their verbalizations simply an "order": whether they received *greater, equal,* or *less* value than they had expected.

To perform this measurement, we coded participants' responses to the retrospective interview questions (Fig. 3). For *expected value* (before they processed the patch), we coded their responses to the "to-method" questions, and to measure their perceived *actual value* of the method (after they processed it), we coded their responses to the "away-from-method" questions. Table 1 shows the code set we used to analyze these navigations.

By these measures, the participants' expectations of the information value they would receive for their foraging efforts were optimistic: they expected Necessary or Sufficient information from about 84% of their navigations (Table 2's top two rows' totals). The first row shows navigations in which participants expected to find *everything* they needed (Sufficient: about 25%), and the second shows navigations in which they expected to find at least *something* they needed (Necessary: about 59%).

"To-method" questions:

- What about location ____ made you go there?
- What did you expect the content to be at location ____?
- Did you consider other options?
- → If yes: What other options did you consider?
- → If yes: Why did the other options not jump out at you, like ____?
- → (If partial list of foraging choices is abandoned) What about these options made you not select any of them?

"Away-from-method" questions:

- Did you find what you expected at location ____?
- → If no: What did you find at location ____?
- What did you learn from location ____?
- Did what you learned cause you to change your course?

Fig. 3. Retrospective semi-structured interview questions.

However, many of these expectations of value were not fulfilled. As Table 2's bottom row shows, 63 of participants' 179 navigations (about 35%) produced lower value than expected. Adding to these disappointments, 28 of the 29 "desperation" navigations (not expected to be either Necessary or Sufficient)—in which participants actually expected no value but tried anyway—indeed led to no actual value. Thus, in total, about 51% of participants' navigations (highlighted cells in Table 2) ended in some degree of disappointment in the information value they received.

Participants rarely found *more* value than they expected. As Table 2's first column shows, participants found higher information value than expected in only 5 navigations, and only one was a participant "lucking into" information in a desperation navigation.

4.1.2 Why: The Challenges of Signposting

To find out why participants' efforts so often returned disappointing value, we analyzed the 63 navigations in which participants received less value than expected, from the perspective of IFT's "cues" construct. What we found was patterns of cues (signposts) that led participants astray in multiple ways.

Many of the words in IDEs refer to places in the code (e.g., method names) and in memory (e.g., variable names), and when they are associated with a clickable or easily scrollable way to navigate to the place to which they refer, these words serve as cues. Because cues like this are identifiable by lexically analyzing source code, we term them *lexical cues*. Participants almost exclusively used lexical cues—mostly method names—to predict the value of a patch to which they were considering navigating.

Unfortunately, this type of cue often misled them to irrelevant patches—even when participants expressed high confidence that the patch would be relevant to their information needs. In particular, three types of problems with lexical cues interfered with the participants' expectations of a patch's value prior to going there: (1) cues that seemed to advertise falsely, (2) synonym cues, and (3) cues answering the "wrong" question.

4.1.2.1 False advertising: Content + where the cue points

Some lexical cues beckoned participants toward a patch with a "false advertisement" of the value. By way of analogy, imagine this sign next to a store window: "Buy <brand name> Coffeemakers". This sign might be just what a shopper needs if the store actually has those coffeemakers, but might lead them astray if it is merely *advertising* the coffeemakers (e.g., sold advertising space). Here the falseness of the advertising lies not in the content of the sign, but the combination of its content and its apparent association with *this* store.

Table 1. Code set for expected (prior to navigating) and actual (after navigating) values.

	Category	Definition
Necessary ⩽ Sufficient	Sufficient	Participants believed that the navigation will ($E(V)$) or did (V) *fully* answer their current foraging goal.
	Necessary	Participants believed the information in the patch at the end of the link will be ($E(V)$) or was (V) necessary & related to their current foraging goal.

Table 2. Participants' expectations of value vs. actual value. Gray cells highlight navigations in which participants had some degree of disappointment (50.8% of navigations).

Expected $E(V)$ \ Actual V	$V > E(V)$	$V=E(V)$	$V<E(V)$	Totals
Necessary and Sufficient	n/a	27 (15.1%)	17 (9.5%)	44 (24.6%)
Necessary, but not Sufficient	4 (2.2%)	56 (31.3%)	46 (25.7%)	106 (59.2%)
Not Necessary, not Sufficient	1 (0.6%)	28 (15.6%)	n/a	29 (16.2%)
Totals	5 (2.8%)	111 (62.0%)	63 (35.2%)	179

At this point, let us briefly consider whether the foundations-oriented perspective we follow in this paper yields useful insights not produced by prior works. For the case of developer navigations, the results produce a new agenda of research challenges, starting with the following:

> *Research Challenge #1 (False Advertising): How to reduce the problem of cues developers interpret as "advertising" prey in a patch that does not, in fact, have that prey.*

The false advertising problem was very common among our participants: P2, P4, P7, P8, and P9 all suffered instances of it. For example, P8 navigated to method `KillRing` because "it's obviously to do with deletion". Yet, upon arriving in the method, he was quickly disappointed when he realized it did not actually perform any of the work of deletion:

> P8 (when asked if he was hoping for something):
> "some more connection to deletion of the actual text... [but] it was just the abstraction"

4.1.2.2 The problem with synonym cues

Some participants used their knowledge of synonyms to navigate. For example, in looking for code that deletes, it seems reasonable to also look for code with names that *mean* the same as "delete". However, synonyms sometimes led our participants astray. P8's `KillRing` false advertising problem above was exacerbated when synonym difficulties also arose. Other examples were:

> P2: "'clear.bsh', is that related to deleting? No it's not"
>
> P7: "I'm assuming 'invalidate' means 'delete' ... Uh, it just doesn't delete"

We were surprised to see the problems that arose with synonyms as cues, because several tools use synonyms directly or indirectly to good effect (e.g., tools powered by natural language vocabulary devices like TF-IDF). For example, the search tool FindConcept uses synonyms to expand the search query [41, 45], and Krec uses standard English synonyms [36].

These approaches bring to mind seminal work on what was originally termed the "vocabulary problem" [14]. That paper showed how huge variations in designers' terminology across numerous application domains are an inherent property of the English language. This result suggests not only the advantage of automatically agglomerating synonyms but also its disadvantage—bringing together synonym-related patches greatly expands developers' search space, as with P2, P6, and P8 above. Thus, too little synonym agglomeration produces too many false negatives, but too much synonym agglomeration produces too many false positives.

> *Research Challenge #2 (Synonyms): How to improve developers' foraging through synonym-filled code without incurring high navigation costs from numerous false positives or false negatives.*

The synonyms problem may relate to Ge et al.'s observation that over 90% of relevant synonyms are unique to software engineering [15]. For example, in software, "invoke" is a synonym of "execute", and "instantiate" is a synonym of "create". They point out that tools could use a thesaurus tailored to the lexicon of SE. Our

results are consistent with this point, but also suggest that the problems with synonym cues may extend beyond that solution.

4.1.2.3 Cues that answered the "wrong" question

Our participants often used lexical cues, such as method names, to try to answer variants of the following foraging question: what will that patch do for my goal? Unfortunately, many of the method names they encountered were never intended to answer that question. Instead, method names generally reflect a method's purpose ("what is this method?"). However, instead of asking "what is" questions, participants often asked "where does" questions, and method names often failed to answer these.

For example, 7 of our 10 participants (P1, P2, P4, P5, P8, P9, P10) ran into trouble foraging for the methods that actually update jEdit's underlying model when a jEdit user performs an editing action. For example, while navigating among numerous method calls in the stack trace, P10 said:

> P10: "I'm trying to figure out which piece of [method] actually *updates* the buffer state"

Variables raised even harder "where does" questions, and here again, lexical cues did not help. For example, P1 was working his way up the exception stack trace, trying to understand where `physicalLine`'s value came from. After several navigations following the execution flow of the program, he finally arrived at a method that did some computations on `physicalLine`.

> P1: "This is the first place where ... there was some *computing* of `physicalLine`, as opposed to just passing it along and throwing exceptions."

Some systems try to address "where does" problems. For example, WhyLine [20] is well-suited for "where does" questions about state and variables, and Reacher [24] answers "where does" questions about methods. However, proof-of-concept tools like these need to be investigated in the context of the entire IDE. Such tools require developers to navigate away from the "main" part of the environment into other tools and screens, potentially causing them to lose context and adding to developers' costs simply by the cost of navigating to other tools.

Table 3. Research challenges with value estimation *E(V)*.

Research Challenge	Participants who encountered it
#1: False advertising (content + where)	P2, P4, P7, P8, P9
#2: Synonym false positives	P2, P7, P8
#3: Cues answering the "wrong" question	P1, P2, P4, P5, P8, P9, P10

Research Challenge #3 (Answering the Wrong Question): How to more often answer the "right" question, i.e., the one a developer is actually asking in their particular situation, given their particular context and state of the IDE.

4.1.3 An Open Problem: The "Value Estimation" Problem with Developers' Navigations

Table 3 summarizes the research challenges in better supporting developers' attempts to predict patch values *before* paying the cost of navigating to those patches. These research challenges come together to reveal a large, open problem space:

> *The Value Estimation Problem (Aligning E(V) with V): How to help developers more accurately predict the value they will gain from planned navigations*—without *bearing the cost of navigating among a plethora of special-purpose tools.*

The challenges identified so far in this paper show that this problem is nuanced, difficult, and multidimensional. Even so, addressing this problem promises high rewards. Recall from Table 2 that solving this problem could potentially improve developers' navigation efficiency by up to 51%.

4.2 RQ2: Developers' Expectations of Cost

4.2.1 Did Participants Incur the Costs Expected?

Participants did not verbalize their expectations of cost before navigating so we did not measure *E(C)* and *C* separately. Instead, we measured how *E(C)* related to *C*, because after navigating they often verbalized a navigation's cost exceeding their expectations ($C > E(C)$). Thus, our code set (Table 4) allocated these verbalizations among the two possible ways costs can be incurred: by navigating between patches (C_b), or by processing within the patch once there (C_w). Thus, $C = C_b + C_w$. The results in Table 4 show that participants discussed facing unexpected costs in 66 of the 179 navigations analyzed (36.9%).

4.2.2 Why: Unanticipated Costs between Patches

Although there were several instances of unexpectedly high within-patch costs C_w (about 13% of the navigations), those can be summarized as simply being time-consuming to understand:

> P1: "Uh, the whole thing was really frustrating. The code was hard to read."
>
> P5: "looking for ... but then I got so lost in [that method], that I didn't really fully understand what was going on."

However, the dominant type of unexpected cost was between-patch, C_b, affecting over 25% of participants' navigations.

Table 4. Frequency of actual costs (C_b or C_w) that were unexpectedly higher than the developers had expected ($E(C_b)$ or $E(C_w)$). (The total is 66 instead of 80 because categories can co-occur in the same navigation.)

Category	Definition	Examples	Navigations affected
Complexity of patch (C_w)	Participants decided that the cognitive difficulty of this patch was unexpectedly high.	Can't understand comments/ documentation. Code too long. Can't figure out what the code is doing.	24 (13.4%)
Surrounding context (C_b)	Participants decided they would now need additional information found only in other patches before they could gain value from this one.	Don't know how to use this code "correctly" without visiting other patches. Don't know what the identifiers represent. Don't know how this code relates to or affects other code.	46 (25.7%)
Time (C_b or C_w)	Participants decided (for unspecified reasons or for reasons other than the above) that the cost of this patch is too high.	Not enough time to process the patch.	10 (5.6%)
		Total:	66 (36.9%)

For between-patch cost C_b, we identified three patterns that the participants faced that repeatedly led to unexpectedly high costs: (1) the prey was in pieces scattered among multiple patches, (2) the path to the prey was long with no end in sight, and (3) sometimes there simply was no available path to the prey.

4.2.2.1 Prey in pieces, scattered among multiple patches

Having prey in pieces spread over multiple patches made foraging costlier than expected because participants had to locate all the relevant patches and assemble the prey themselves. Using the coffeemaker analogy from before, this would be like buying a coffeemaker in parts, with a different store exclusively selling each part. To get a working coffeemaker, one would have to go to one store to get a handle, another to get the glass container, yet another to get a lid, and so on, only to then also assemble the gathered components before brewing any coffee.

> *Research Challenge #4 (Prey in Pieces): How to better support developers who are having to assemble prey that is in pieces scattered among multiple patches.*

One situation in which participants had to collect and assemble information from multiple patches was when they tried to learn semantic information, such as what a variable represented. For example, P1, P3, P6, P7, and P9 were confused by all of the different *line* variables. Documentation for the semantic differences between the variables existed within comments in the code, but participants sometimes needed a combination of knowledge from several patches to understand the different variables.

> P5: [Did you consider any other choices besides <method name>?] "No, because the names didn't really say much to me ... I basically had no clue about *context*..."

To illustrate the cost involved in establishing such context, consider P1's case. As Fig. 4 shows, for P1 to build the context he wanted, he would have had to not only move up the call stack to find the relevant relationships and documentation, but also had to locate and navigate through the call relationships through the dashed methods in the figure before putting together his desired prey. Of course, P1 had no way of knowing this, and after foraging within the first four methods of the call stack, he gave up.

Several other participants faced similar difficulties when they wanted to understand where and how the value of a variable changed during execution. P1, P3, P6, and P10 all navigated between several methods that executed during the Delete Line action to determine how specific values of variables changed. One example was the `physicalLine` variable. To understand where `physicalLine` came from, participants navigated up the call hierarchy and identified patches where `physicalLine` was being modified only to reach a method that showed that `physicalLine` was calculated using a `screenLine` as a parameter. Then they had to navigate through another call hierarchy. With each additional variable, there was yet another call hierarchy to investigate, and the number of patches to investigate grew rapidly. Eventually, all four of these participants decided the cost was too high, and gave up.

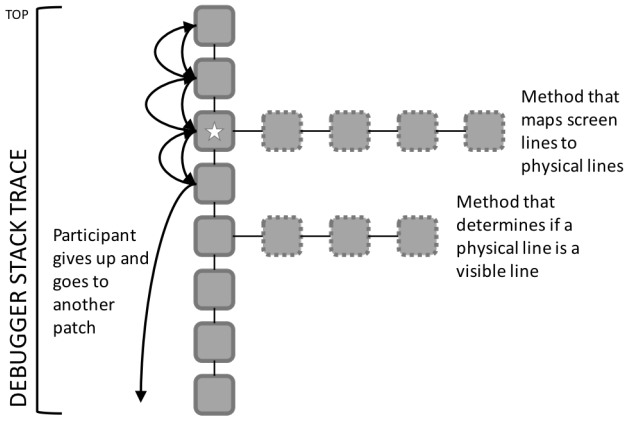

Fig. 4. P1 was looking for the relationship between screen lines and visible lines, after seeing both in the starred method. But jEdit has three line types, so he would have *also* needed to understand physical lines from the dashed locations (far right).

4.2.2.2 The path to the prey is too long, with no end in sight

In contrast to the above, with the prey being scattered about in pieces, some participants' prey were already fully assembled and in only one patch—but the path was so long, participants thought they were going in the wrong direction and gave up.

Returning to the coffeemaker analogy, imagine entering a store searching for a coffeemaker, but having the clerk tell you they do not sell them, but they can point you to a store that might. Then, upon entering that store, having that clerk send you to yet another store. Eventually, you might get to a store with the coffeemaker, or you might give up before you get there because with each trip to a new store, it seems less and less likely that any of the stores has a coffeemaker available. Several participants engaged in this behavior of going from patch to patch, until finally giving up.

> *Research Challenge #5 (Endless Paths): How to better support developers when the path to the prey is very long, so that the developer does not erroneously decide that the prey is not on that path.*

For example, when P3 was looking for methods related to folding or deleting text, he set a breakpoint in the exception-throwing method that he identified earlier. He then foraged through the sequence of methods in the debugger's stack frames working his way down the stack, sometimes returning to a previous frame to regain lost context. After several navigations, he gave up—still three methods away from the `deleteLine` method that he was looking for (Fig. 5). P1, P2, P3, P4, P5, P8, and P9 all experienced this expense of navigating through long sequences of patches en route to their desired prey.

4.2.2.3 Sometimes there is no path

Some participants could not find a path to their prey because the information they wanted was located in a different topology altogether. A *topology* is a collection of patches and the links between them. In this study, one topology was the code itself, with units of code (such as methods or classes) being the patches and the ways to navigate between them (like scrolling or using various IDE

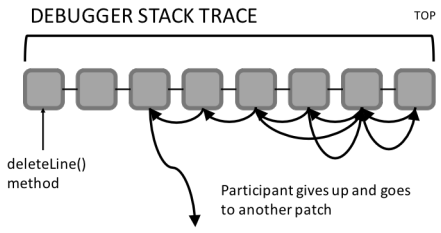

Fig. 5. P3 navigated down the debugger's stack frames searching for methods related to folding or deletion. After several navigations down the stack, he gave up only three methods away from the method he was looking for.

navigation affordances) being the links. Another topology, disjoint from the code topology, was the jEdit running instance, with its own patches and navigation affordances not connected to code. In the Eclipse IDE, participants sometimes formulated a foraging goal while in one topology, but had to fulfill the goal in another. What was missing was a way for participants to easily navigate between related patches from one topology to another.

> *Research Challenge #6 (Disjoint Topologies): How to enable developers to navigate through related patches among multiple, disjoint topologies.*

This inability to move between topologies in a low-cost way primarily manifested when participants were mapping runtime GUI behavior to code. For example, several participants, while recreating the bug in jEdit's running instance, formulated the goal of finding the code that was triggered by GUI actions. However, after formulating that goal, they then had to switch over to jEdit's code and start a fresh set of navigations, since there was no direct way to go from executing the action in jEdit to the code that handles that action. Instead, participants located the relevant code by using search tools, by investigating the stack trace, or by setting a breakpoint and stepping through code. Fig. 6 shows one common missing link between the two topologies, mapping the Delete Lines menu action to the `deleteLine` method.

In the above situation, participants had to resort to finding the relevant representations of GUI elements in the code by navigating through code, which was both costly and unfruitful. P3 set a breakpoint and then navigated through several frames trying to find information. P4 chose to trigger an action related to the bug and step through several methods of code to find relevant prey. P5 simply selected a relevant-looking method from the outline view and started to read code.

For jEdit, there were four disjoint topologies: the GUI runtime, the source code, the external menu library, and the XML properties file. Besides the GUI runtime topology and source code topology, jEdit uses an external library—the third topology—to automatically build menus based on the content of an XML properties file—the fourth topology disjoint from the others. The properties file defined the content of the menus and specified which methods to fire for each menu item. The disjointness was a source of confusion for participants, since many of the methods that were called by menu items had no callers when an open call hierarchy action was used in Eclipse.

There are a few beginnings toward addressing the disjoint topologies challenge. For example, Whyline [20] builds a path just-in-time to bridge the gap between two topologies: from GUI output to its relevant source code, and SketchLink [3] links sketches to code. Mining approaches like Chen and Grundy's [5] are also emerging to find relationships among disjoint topologies such as documentation and source code. However, Whyline does not scale to a program of jEdit's size, approaches like Chen/Grundy's do not support navigations per se, and few of the approaches we have located handle more than two disjoint topologies. Still, these approaches provide promising starts upon which to build.

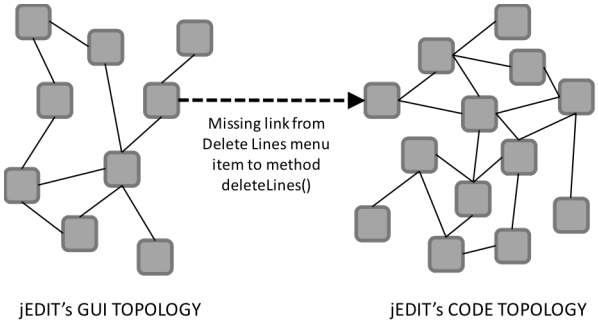

Fig. 6. P7 said he wanted to navigate from the Delete Lines menu action to the `deleteLines` code, but the environment had no link from the action to the code it triggered. (Solid lines: links present. Dashed line: missing link.)

4.2.3 An Open Problem: The "Cost Estimation" Problem with Developers' Navigations

At Table 4 showed, about 37% of developers' costs were much higher than they had expected. In essence, developers had to navigate to patches without knowing what it would cost until *after* they had paid—a situation not unlike writing a blank check for the coffeemaker of our earlier analogy. Table 5's summary of cost-related foraging research challenges contributing to these issues reveals a substantive and difficult open problem space analogous to the Value Estimation Problem presented in Section 4.1:

> *The Cost Estimation Problem (Aligning E(C) with C): How to enable developers to more accurately predict the foraging costs they will incur* before *they incur them.*

5. RQ3: LITERATURE ANALYSIS

To answer our third research question, whether recent trends in software engineering research have begun to address these problems, we conducted a literature analysis of 302 papers from three literature repositories. The first repository was the 99 papers cited in the most recent (2013) journal paper surveying SE tools that contribute to developers' information foraging [12], which included, for example, tools helping collect information for debugging, reuse, or infering what a developer seeks, and for recommending appropriate resources (e.g., [6, 20, 36, 38]). The 2013 journal paper [12] sampled literature from a wide range of dates, so to ensure currency, we added two very recent repositories. Thus, the second repository was FSE'14 (104 papers), which was the most recent FSE available at the time we began this analysis, and the third was ICSE'14 (99 papers), i.e., the same year as the FSE repository.

5.1 Analysis Methodology

From the resulting 302 papers, we selected for detailed analysis all papers that met the following criteria: (1) it must describe a tool that supports a software engineering foraging activity, (2) the activity must have a before-navigation and an after-navigation state, and (3) the paper (or related resources) must include information of the navigation choices a developer can make.

We then qualitatively coded the 55 papers that met these criteria based on the description of the foraging activity supported by the paper's tool (or by following references in the paper to other resources describing the tool), using the code set given in Table 6. As the table shows, the codes cover every possible way to align value V with $E(V)$ if $V<E(V)$, and to align cost C with $E(C)$ if $C>E(C)$ for the two factors of C, namely C_b and C_w.

Table 5. Research challenges for cost estimation $E(C)$.

Research Challenge	Participants who encountered it
#4: Prey in pieces scattered among several patches	P1, P3, P4, P5, P6, P7, P9, P10
#5: Path too long, no end in sight	P1, P2, P3, P4, P5, P8, P9
#6: No path across different topologies	P1, P2, P3, P4, P5, P6, P7, P8, P9, P10

To ensure reliability of our analysis, we followed the same inter-rater reliability (IRR) practices we described for the other code sets in this paper. Specifically, two researchers independently coded the same 20% of the data, and calculated their level of agreement using the Jaccard index. After achieving 90% inter-rater reliability on the first repository and 81% on the remaining two, they divided up the coding of the remaining data.

5.2 Results

Table 7 presents the results of our analysis of the 55 SE research tools. As per the underlying code set (Table 6), Table 7 has a column for every possible way a tool could improve developers' mismatches in actual versus expected value or cost, and shadings show which tools contributed to each.

A visual scan of the shaded cells in Table 7's columns 3–8 reveals four results. The first is good news regarding SE research's commitment to enhancing the value and cost of developers' information seeking—100% of the 55 tools make some kind of contribution to helping developers with aspects of value or cost.

5.2.1 Improving the Actuals: V and C

The second result, shown by Table 7's columns 3–5, is that most of these tools (47/55=85%) are working toward improving developers' *actual* value V or cost C.

More specifically, Table 7's column 3 ("Increase V") shows that just over half of these 47 papers (26) work toward increasing the value V a patch delivers to developers who make their way there. These tools do so by *adding information features* to that patch.

One example is SketchLink [3], which adds sketch diagrams relevant to the current method (Figure 7)—which increases V provided that these added information features help to answer the question(s) the developer actually had, as per Research Challenge #3 (answering the wrong question). When that provision is met, a best case is that value V to a developer might increase from necessary up to sufficient. This best-case increase could help with Table 2's result that developers' navigations did not usually produce value that was sufficient.

Turning to the next two columns, work to reduce costs dominates the "actuals"—47 papers contribute toward decreasing C_b and/or C_w. All except one of these 47 focuses on decreasing cost C_b of navigating to a patch, but over half (29) focus also (or in one case, instead) on the cost C_w of navigating within that patch.

For example, SketchLink [3] (Fig. 7) reduces C_w by *making explicit* information the developer would otherwise need to infer by studying the code. It also reduces C_b by *adding links* between two disjoint topologies, sketches and code, as per the two-topology case of Research Challenge #6 (disjoint topologies).

5.2.2 Aligning Expectations: E(V) and E(C)

Improving actual V and C is important, but it still leaves an important gap—it does not resolve the waste than ensues if developers cannot *predict in advance* whether they will receive value until after they pay the cost. This is why aligning *E(V)* and *E(C)* with V and C, respectively, matters.

The third result is about this alignment. At first, Table 7 gives an impression visually that aligning *E(V)* with V is very common among these tools, with 52/55 (95%) making some effort to do so. However, this impression is a bit misleading, because most approaches help developers predict patch values *only for patches that are nearby* (one navigation away), a point we shall return to shortly. Still, one excellent example of support of *E(V)* is Team Tracks [9], which helps developers predict value by rating patches according to how often the developer's team visited them (Figure 8, left), regardless of how many navigations away the patch is.

Table 6. Code set for the literature analysis. *E*(*V*) = expected value, *E*(*C*) = expected cost. *V* = actual value, *C* = actual cost.

	Code	Description
Align of expected	Aligns accuracy of $E(V)$ with V.	Prior to a navigation, a cue hints at the value of information at the end of the link.
	Aligns accuracy of $E(C_b)$ with C_b of navigating *between* patches.	Prior to a navigation, a link gives clues (via the cues) as to the cost of navigating to the patch at the end of the link.
	Aligns accuracy of $E(C_w)$ with C_w of processing *within* a patch.	Prior to a navigation, a link gives clues (via the cues) as to the cost of processing a patch (e.g., context, complexity, time).
Improve actual	Increases V of a patch	The patch has been modified to increase its value, either by adding relevant information or removing irrelevant information.
	Decreases C_b of between-patch foraging	Developers can navigate to a desired patch more quickly.
	Decreases C_w of within-patch processing	After a navigation, the patch itself has been modified to decrease its processing costs, either through the removal of irrelevant information features or by drawing attention to relevant information features.

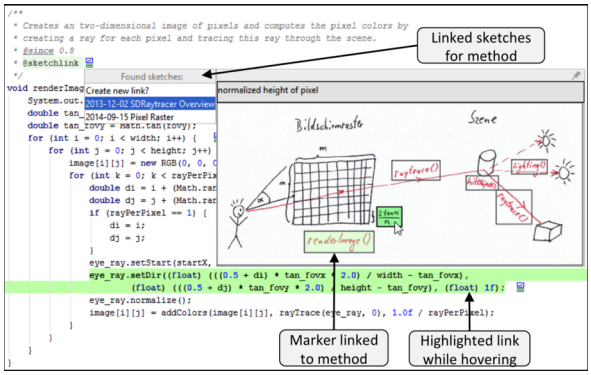

Fig 7. SketchLink improves *V*, C_b, & C_w in overlapping ways: Increasing *V:* It adds information (the floating sketch) about the current method.
Decreasing C_b: Developers can get the sketch without a tedious sequence of navigations.
Decreasing C_w: Developers need not study the code to infer information the sketch makes explicit.

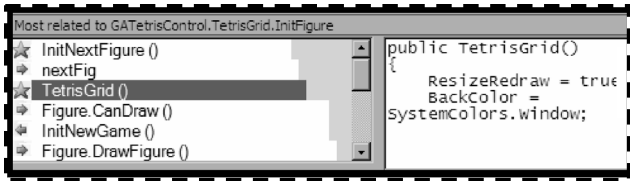

Fig 8. Team Tracks' support for both *E*(*V*) and *E*(C_w): (Left) Aligning *E*(*V*) with *V:* Shows how often fellow team members visited it as an estimate of value *V*.
(Right): Aligning *E*(C_w) with C_w: If a developer selects a method, it shows a preview, helping developers predict how long they will spend understanding the method.

Table 7. Results of analyzing 302 papers, showing the 55 tools that assist developers with tasks involving foraging. Shaded = some support, blank=none, ? = unclear in the paper. Citations are in ICSE'14/FSE'14 if indicated; otherwise, see [12] for full citations.

Paper	Supports what foraging goal	Increase V	Decrease C_b	Decrease C_w	Align V & $E(V)$	Align C_b & $E(C_b)$	Align C_w & $E(C_w)$
Alimadadi et al. … JavaScript … ICSE 2014	Understand interaction between source code components.	■	■	■	■		■
Alves et al., RefDistiller… FSE 2014	Locate potential errors caused by manual refactoring.	■	■	■	■		
Ashok et al. DebugAdvisor …2009	Locate relevant information associated with a bug.		■		■		
Baltes et al. Linking sketches … FSE 2014	Locate/link sketches/diagrams relevant to part of the code.	■	■	■	■		
Bragdon et al. Code Bubbles … 2010	Locate and visually organize code.	■	■	■	■		
Caldiera et al. … reusable software… 1991	Locate reusable code.	?	?	?	■	?	■
Coblenz et al. JASPER… 2006	Collect relevant code in a separate, easy-to-navigate patch.		■		■		
Cottrell et al. …source code reuse…. 2008	Locate reusable code.	■	■	■	■		
Cubranic et al. HIPIKAT… 2005	Locate artifacts relevant to the current context.	■	■	■	■		
De Alwis et al. …Ferret. 2008	Locate and collect code based on developer's query.		■		■	■	
DeLine et al. … sharing navigation data. 2005	Find relevant code via team members' navigation histories.	■	■	■	■		■
Duala-Ekoko et al. … code clones… 2007	Locate and collect code clones for modification.	■	■	■	■		■
Ducasse et al. … duplicated code. 1999	Locate duplicated code.		■		■		
Dudziak. … structural weaknesses …. 2002	Locate potential bad code via code smells.	■	■	■	■		
Dunn et al. … reusable parts... 1993	Locate reusable code.	?	?	?	■	?	?
Fritz et al. information fragments … 2010	Locate/organize code collaborated on by several developers.		■		■		
Galenson et al. CodeHint… ICSE 2014	Find code snippets that meet the requirement/specification.	■	■	■	■	■	
Ge et al. Manual refactoring … ICSE 2014	Locate and/or fix errors caused by automated refactoring.	■	■	■	■	■	
Henninger. … find reusable software. 1994	Locate reusable code.		■		■		
Hermans et al. BumbleBee… FSE 2014	Locate source code files that may contain the bug.	■	■	■	■		
Hill et al. NL-queries for software … 2009	Locate relevant program elements based on NL-queries.				■		
Holmes et al. … context matching… 2006	Locate examples for a particular source code element.		■	■	■		■
Holmes et al. … pragmatic reuse tasks. 2007	Organize and annotate code fragments during reuse tasks.		■		■		
Kaleeswaran et al. MintHint… ICSE 2014	Identify changes needed to a fix a buggy piece of code.		■	■	■		■
Kersten et al. …task context … 2006.	Collect relevant code in a separate, easy-to-navigate patch.		■	■	■		
Ko. Asking and answering questions… 2008	Locate code that caused a particular output.	■	■	■	■		
Lanubile et al. Function recovery … 1993	Reverse-engineer components.		■		?		
Layman. Information needs … 2009	Identifies code relationships for a given code element.	■	■		■		
Lin et al. Detecting differences… ICSE 2014	Locate clones and identify similarities and differences.	■	■	■	■		■
Manotas et al. SEEDS… ICSE 2014	Identify possible changes for more energy-efficient code.			■	■		■
McMillan et al. Exemplar…2012	Locate relevant software projects based on NL-query.		■		■		
Mens et al. Beyond the refactoring … 2003	Locate potential bad code via code smells.	■	■	■	■		
Minto et al.. … emergent teams. 2007	Locate an expert for a given section of code.		■	■	■		■
Mirakhorli et al, Archie… FSE 2014	Find code that matches a certain architectural pattern.	■	■	■	■		
Mockus et al.. Expertise browser… 2002	Locate an expert for a given section of code.		■		■		
Ocariza et al. Vejovis… ICSE 2014	Identify possible changes to fix a buggy piece of code.		■	■	■		
Okur et al. … in C#. ICSE 2014	Identify fixes to bugs with asynch. programming constructs.	■	■	■	■		
Olivero et al. … object-focused … 2011	Locate and visually organize code.	■	■		■		
Parnin et al. … usage contexts … 2006	Locate and recommend context-relevant code.		■		■		
Reiss. Semantics-based code search. 2009	Locate code based on developer-supplied specification.		■		■		
Schiller et al. … specifications. ICSE 2014	Identify formal behavioral specifications & document them.	■	■	■	■	■	
Simon et al. Metrics based refactoring. 2001	Locate code suitable for refactoring via code smells.				■		
Storey et al. … waypoints… 2007	Locate code tagged with user-determined categories.	■	■		■		
Subramanian et al. Live API … ICSE 2014	Understand what code does via documentation & examples.	■	■		■		
Thung et al., BugLocalizer… FSE 2014	Locate the files that may potentially contain the bug.				■		
Tokuda et al. … object-oriented … 2001	Locate code for refactoring.	■	■	■			
Toomim et al. Managing duplicated … 2004.	Locate code duplicates.	■	■	■			■
van Emden et al. Java quality … 2002	Locate bad-smelling code.				■		
Wursch et al. … natural language 2010	Locate code based on NL-queries.		■		■		
Xiao et al. Titan… FSE 2014	Find relationships between classes.				■		
Ye, Y et al. … active information … 2000	Locate code for reuse.	■	■	■	■		
Ye, Y et al. A socio-technical … 2007	Locate an expert for a given section of code.	■	■	■	■		
Zhang et al. … configuration … ICSE 2014	Locate/fix configuration-related errors in a newer version.				■		
Zhang et al. Critics… FSE 2014	Identify code where similar systematic changes occurred.		■	■	■		
Zimmermann et al. Mining version … 2004	Recommend code needing modification.		■		■		

The fourth result Table 7 reveals is that tools helping to align $E(C)$ with C were relatively rare—only 4/55 (7%) made any attempt to align $E(C_b)$ with C_b, and only 10/55 (18%) worked to align $E(C_w)$ with C_w. As an example of supporting $E(C_w)$, when developers using Team Tracks select an item in the list Team Tracks recommends, it shows a preview (Figure 8, right), to help developers predict the cost of understanding the code. However, as with the $E(V)$ work, few tools handle patches that reside more than one navigation away.

5.3 The Scaling Up Problem

Section 5.2's examples provide useful ideas toward ultimately addressing some of the research challenges of Section 4, but only a few help the developer align $E(V)$ with V or $E(C)$ with C for patches more than one click away. This leaves the developer in a state of acute myopia (near-sightedness), unable to see beyond one navigation away—and thus unsupported in coping with long-distance problems such as those illustrated by Figures 4 and 5.

This suggests our third and final open problem space:

> *The Scaling Up Problem (More than one click away): How to enable developers to accurately predict value/cost of multiple "distant" patches (i.e., more than one navigation away).*

Fortunately, our literature analysis points to a few notable starts in this direction. Besides the examples above, other examples are MCIDiff [28] and CloneTracker [11]. These two clone-tracking tools consider the possible set of clones, show the length and the number of instances of *all* code clones, not just nearby ones, to help the developer predict $E(C_w)$ of handing any or all of these clones. Another useful example is the query system Ferret [7], which helps developers predict $E(C_w)$ beyond the one-click-away distance. Ferret allows developers to ask conceptual queries about a particular program element such as "What methods instantiate this type?" and while displaying the results, also shows the number of results for the query, thus allowing the developer to gain some idea of the sum of all the $E(C_w)$s they will incur. Promising starts like these works can serve as the ground floor upon which SE researchers can build toward ultimately addressing this problem.

6. THREATS TO VALIDITY

Every study has threats to validity. We guarded against threats to internal validity in several ways. For our empirical study, our inter-rater reliability was 81%–92% on all code sets, helping to assure construct validity (the extent to which a measure actually captures what is intended). To further help assure construct validity, we did not rely on our raters' interpretations alone, but also used retrospective interviews to remind participants where they were and also to gather participants' interpretations of which events mattered and why they did what they did. However, participants' recollections of what happened later may have biased their responses.

The primary threat to external validity of the empirical study is that our study participants were new to the code base. This is a common situation for new hires and when developers transfer to different development teams, but we do not expect it to generalize to other kinds of debugging situations. Additionally, there is a question of generalizability to other programming languages and IDEs, which we defer to future work.

For our literature analysis, we also had good inter-rater reliability, with 81%–90% on 20% of the data. However, a threat to external validity is that our sampling of current SE trends may not be complete. Our analysis covered 302 papers from top publication venues, but these SE literature repositories still may not generalize across all SE literature.

Table 8: Summary of open research problems and challenges.

Open Research Problems and Challenges	Described in...
The Value Estimation Problem	Section 4.1.3
Research Challenge #1: False advertising	Section 4.1.2.1
Research Challenge #2: Synonyms	Section 4.1.2.2
Research Challenge #3: Answering "wrong" question	Section 4.1.2.3
The Cost Estimation Problem	Section 4.2.3
Research Challenge #4: Prey in pieces	Section 4.2.2.1
Research Challenge #5: Endless paths	Section 4.2.2.2
Research Challenge #6: Disjoint topologies	Section 4.2.2.3
The Scaling Up Problem	Section 5.3

7. CONCLUDING REMARKS

In this paper, we used an Information Foraging Theory perspective to investigate developers' navigation decisions, how often these decisions led to disappointment, and the fundamentals of why. The results suggest that how well a developer can *predict* the value and/or cost of a navigation path are critical factors of the lower bound of a developer's navigation efficiency in a given information space. The results further suggest a new area of inquiry for SE researchers: how large, feature-dense SE environments can support developers' ability to predict the value they will receive from a navigation path and the cost they must expend to receive that value. As Sections 4–5 showed and Table 8 summarizes, this area of inquiry appears to be rich, challenging, and cross-cutting, with three open problem spaces involving (at least) six research challenges.

Our empirical study showed these problems to have significant effects on developers' productivity, with high percentages of expensive wasted developer effort. For example, about 51% of the developers' foraging decisions led to disappointing value of information obtained, and about 37% of the developers' foraging decisions resulted in higher than anticipated costs. Conservatively assuming that all value and cost foraging disappointments overlapped, about 51% of their foraging resulted in disappointment; a worst-case summation that assumes no overlap in these disappointments is 88% of their navigations leading to disappointment.

Further, our literature analysis revealed only a little evidence of SE research tools that aim squarely at helping developers to align their *predicted* navigation value and cost with the actual values and costs they will incur. Fortunately, a few such tools make useful inroads in directions needed to help address the issues, as we pointed out in the literature analysis.

Together, these results are a call for action. Developers' future productivity will depend on SE researchers' ability to make significant progress toward solving the open problems and challenges that were revealed by considering developer navigation at a foundational level.

> P8: "... really hard … it's just, you know, miles of methods, miles of methods."

8. ACKNOWLEDGMENTS

This work was supported in part by Oracle, by NSF under Grants 1302113, 1302117, and 1314384, and by David Piorkowski's IBM PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Aho, A., Lam, M., Sethi, R., and Ullman, J. 2006. *Compilers: Principles, Techniques & Tools*. Addison Wesley.

[2] Bajracharya, S. K. and Lopes, C. V. 2012. Analyzing and mining a code search engine usage log. *Empirical Soft. Engr.*, *17*(4-5), 424–466.

[3] Baltes, S., Schmitz, P., and Diehl, S. 2014. Linking sketches and diagrams to source code artifacts. *ACM Symp. Found. Soft. Engr. (FSE)*, 743–746.

[4] Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *Intl. J. Man-Machine Studies*, *18*(6), 543–554.

[5] Chen, X. and Grundy, J. 2011. Improving automated documentation to code traceability by combining retrieval techniques. *IEEE/ACM Intl. Conf. Automated Software Engineering (ASE)*, 223–232.

[6] Cottrell, R., Walker, R., and Denzinger, J. 2008. Semi-automating small-scale source code reuse via structural correspondence. *ACM Symp. Found. Soft. Engr. (FSE)*. 214–225.

[7] De Alwis, B. and Murphy, G. C. 2008. Answering conceptual queries with Ferret. *ACM/IEEE Int'l. Conf. Software Engineering (ICSE)*. 21–30.

[8] DeLine, R., Khella, A., Czerwinski, M. and Robertson, G. 2005. Towards understanding programs through wear-based filtering. *ACM Symp. Software Visualization (SoftVis)*, ACM, 183–192.

[9] DeLine, R., Czerwinski, M., and Robertson, G. 2005. Easing program comprehension by sharing navigation data. *IEEE Symp. Visual Languages and Human-Centered Computing (VL/HCC)*. 241–248.

[10] Duala-Ekoko, E. and Robillard, M. P. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE),* 266–276.

[11] Duala-Ekoko, E. and Robillard, M. 2007. Tracking code clones in evolving software. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*. 158–167.

[12] Fleming, S., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. 2013. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans. Soft. Engr. and Method. (TOSEM)*, 22(2), 14:1–14:41.

[13] Fritz, T., Shepherd, D. C., Kevic, K., Snipes, W., and Bräunlich, C. 2014. Developers' code context models for change tasks. *ACM Intl. Symp. Found. Soft. Engr. (FSE)*, 7–18.

[14] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. 1987. The vocabulary problem in human-system communication. *Comm. ACM, 30(11)*, 964–971.

[15] Ge, X. and Murphy-Hill, E. 2014. Manual refactoring changes with automated refactoring validation. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*, 1095–1105.

[16] Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., and Kwan, I. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Trans. Comp.-Human Interaction (TOCHI)*, 19(1), 5:1–5:28.

[17] Henley, A. and Fleming, S. 2014. The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. *ACM Conf. Human Factors in Comp. Sys. (CHI).*, 2511–2520.

[18] Karrer, T., Kramer, J., Diehl, J., Hartmann, B., and Borchers, J. 2011. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. *ACM Symp. User Interface Soft. and Technology (UIST)*, 217–224.

[19] Kevic, K., Fritz, T., and Shepherd, D. 2014. CoMoGen: An approach to locate relevant task context by combining search and navigation. *ACM/IEEE Intl. Conf. Soft. Maint. and Evolution (ICSME)*, 61–70.

[20] Ko, A. and Myers, B. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. *ACM/IEEE Int'l Conf. Software Eng. (ICSE)*, 301–310.

[21] Ko, A., Myers B., Coblenz, M., Aung, H. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Soft. Eng (TSE). 33*, 971–987.

[22] Kramer, J., Karrer, T., Kurz, J., Wittenhagen, M., and Borchers, J. 2013. How tools in IDEs shape developers' navigation behavior. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 3073–3082.

[23] LaToza, T. D. and Myers, B. A. 2010. Developers ask reachability questions. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*, 185–194.

[24] LaToza, T. D. and Myers, B. A. 2011. Visualizing call graphs. *IEEE Symp. Visual Lang. and Human-Centric Computing (VL/HCC)*, 117–124.

[25] Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 1323–1332.

[26] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S. 2013. How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Soft. Eng. (TSE)*, 39(2), 197–215.

[27] Lawrance, J., Burnett, M., Bellamy, R., Bogart, C., and Swart, C. 2010. Reactive information foraging for evolving goals. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 25–34.

[28] Lin, Y., Xing, Z., Xue, Y., Liu, Y., Peng, X., Sun, J., and Zhao, W. 2014. Detecting differences across multiple instances of code clones. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*, 164–174.

[29] Maalej, W., Tiarks, R., Roehm, T., and Koschke, R. 2014. On the comprehension of program comprehension. *ACM Trans. Soft. Engr. and Method. (TOSEM)*, 23(4), 31:1–31:38.

[30] Majid, I. and Robillard, M. 2005. NaCIN: An Eclipse plug-in for program navigation-based concern inference. *ACM OOPSLA Wkshp. Eclipse Technology Exchange*, 70–74.

[31] Niu, N., Mahmoud, A., Chen, Z. and Bradshaw, G. 2013. Departures from optimality: Understanding human analyst's information foraging in assisted requirements tracing. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*, 572–581.

[32] Piorkowski, D., Fleming, S., Kwan, I., Burnett, M., Scaffidi, C., Bellamy, R., and Jordahl, J. 2013. The whats and hows of programmers' foraging diets. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 3063–3072.

[33] Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R., and Swart, C. 2012. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 1471–1480.

[34] Pirolli, P. and Card, S. 1995. Information foraging in information access environments. *ACM Conf. Human Factors in Comp. Sys. (CHI)*, 51–58.

[35] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. 2013. Dual ecological measures of focus in software development. *ACM/IEEE Int'l Conf. Software Eng. (ICSE)*, 452–461.

[36] Robillard, M. P. and Chhetri, Y. B. 2014. Recommending reference API documentation. *Empirical Soft. Engr.*, 1–29.

[37] Robillard, M., Coelho, W., and Murphy, G. 2004. How effective developers investigate source code: An exploratory study. *IEEE Trans. Soft. Eng. (TSE),* 30(12), 889–903.

[38] Sawadsky, N., Murphy, G. C., and Jiresal, R. 2013. Reverb: Recommending code-related web pages. *ACM/IEEE Intl. Conf. Soft. Engr. (ICSE)*, 812–821.

[39] Seaman, C.B. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Soft. Engr. (TSE*), 25(4), 557–572.

[40] Shaw, M. 1990. Prospects for an engineering discipline of software, *IEEE Software,* 15–24.

[41] Shepard, D., Fry, Z. P., Hill, E., Pollock, L. and Vijay-Shanker, V. 2007. Using natural language program analysis to locate and understand action-oriented concerns. *ACM Int'l Conf. Aspect-Oriented Soft. Dev. (AOSD)*. 212–214.

[42] Sillito, J., Murphy, G. C., and De Volder, K. 2006. Questions programmers ask during software evolution tasks. *ACM Symp. Found. Soft. Engr. (FSE)*, 23–34.

[43] Singer, J., Elves, R. and Storey, M. 2005. NavTracks: Supporting navigation in software maintenance. *IEEE Intl. Conf. Soft. Maint. (ICSM)*, 325–334.

[44] Soh, Z., Khomh, F., Gueheneuc, Y., Antoniol, G., and Adams, B. 2013. On the effect of program exploration on maintenance tasks. *Working Conf. Reverse Engr.*, 391–400.

[45] Sridhara, G., Hill, E., Pollock, L. and Vijay-Shanker, K. 2008. Identifying word relations in software: A comparative study of semantic similarity tools. *IEEE Int'l Conf. Prog. Comprehension (ICPC)*, 123–132.

[46] Xuan, Q., Okano, A., Devanbu, P., and Filkov, V. 2014. Focus-shifting patterns of OSS developers and their congruence with call graphs. *ACM Int'l. Symp. Found. Soft. Engr. (FSE)*, 401–412.

[47] Ying, A. and Robillard, M. 2011. The influence of the task on programmer behaviour. *IEEE Int'l Conf. Program Comprehension (ICPC),* 31–40.