

The Whats and Hows of Programmers' Foraging Diets

David Piorkowski¹, Scott D. Fleming², Irwin Kwan¹, Margaret Burnett¹,
Chris Scaffidi¹, Rachel Bellamy³, Joshua Jordhal¹

¹Oregon State University
Corvallis, Oregon, USA

²University of Memphis
Memphis, Tennessee, USA

³IBM Research
Hawthorne, New York, USA

{piorkoda, kwan, burnett, cscaffid}@eecs.oregonstate.edu,
scott.fleming@cs.umemphis.edu, rachel@us.ibm.com

ABSTRACT

One of the least studied areas of Information Foraging Theory is diet: the information foragers choose to seek. For example, do foragers choose solely based on cost, or do they stubbornly pursue certain diets regardless of cost? Do their debugging strategies vary with their diets? To investigate “what” and “how” questions like these for the domain of software debugging, we qualitatively analyzed 9 professional developers’ foraging goals, goal patterns, and strategies. Participants spent 50% of their time foraging. Of their foraging, 58% fell into distinct dietary patterns—mostly in patterns not previously discussed in the literature. In general, programmers’ foraging strategies leaned more heavily toward enrichment than we expected, but different strategies aligned with different goal types. These and our other findings help fill the gap as to what programmers’ dietary goals are and how their strategies relate to those goals.

Author Keywords

Information foraging theory; information diet; debugging strategies

ACM Classification Keywords

D.2.5 [Software Engineering]: Testing and Debugging;

H.1.2 [Information Systems]: User/Machine Systems—Human factors

INTRODUCTION

Pirolli et al.’s pioneering work on Information Foraging Theory (IFT) [18] has greatly influenced our community’s understanding of how humans seek information within information-rich environments such as the Web. The theory is based on the idea that humans seek information in a manner analogous to the way animals seek food in the wild. In short, it states that a human information predator seeks information prey by following information scent through an environment. IFT has been well validated empirically (e.g., [3,6,12,13,14,20]). It has facilitated predictive models of how people navigate as they forage within websites (e.g.,

[3,18,21]) and during software maintenance tasks (e.g., [12,13]). Furthermore, the theory has spawned principles for the design of interfaces and tools that help people forage (e.g., [16,24]).

One area of potential for IFT that so far has been mostly untapped is using the theory to understand the *diets* of predators in a particular problem domain—that is, to understand the types of information goals those predators desire. A notable exception is Evans and Card [4], who investigated the diets of web users who were “early adopters.” They discovered that these users’ diets were considerably different from the information commonly provided by mainstream news sites, and they identified the niche topics that made up the users’ diets. They also noted that the information sources chosen by these users reduced the cost of attention by lowering the cost of social foraging and social interpretation. Clearly, these findings have strong implications for the design of sites to support such users. The Evans and Card work demonstrates the potential benefits of applying information foraging ideas to understand the diets of people in particular contexts.

Inspired in part by the Evans/Card paper, our work aims to expand our understanding of IFT diets by investigating the diets of professional software developers engaged in debugging. Work in the software engineering (SE) literature has investigated related ideas, such as the questions that programmers ask (e.g., [5,10,11,23]), but that work was not grounded in a theory, such as IFT. Thus, by investigating the information diets of professional programmers from an IFT perspective, our work aims to help bridge the gap between such results from the SE literature and the IFT foundations and results from the HCI literature.

For an understanding of the “whats” of diet to be truly useful, we also need to understand the “hows”. Toward this end, we also investigate, from an IFT perspective, the *strategies* that programmers use during foraging. The literature contains numerous works on program debugging strategies (see [22] for a summary), but these have not been tied to IFT. We believe that such strategies both influence and are influenced by programmers’ diets, and this paper investigates these ties.

Thus, in this paper, we address the following research questions with a qualitative empirical study.

This tech report is an extended version of publication [17]: It adds Appendix A to the end.

© ACM, 2013. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in [17].

- RQ1 (diet “whats”): What types of information goals do professional programmers forage for during debugging, and how do those goals relate to one another?
- RQ2 (foraging “hows”): How do professional programmers forage: what foraging strategies do they use?
- RQ3 (“whats” meet “hows”): Do professional programmers favor different strategies when foraging for different types of information?

BACKGROUND

Information Foraging Theory

Information foraging theory is a theory of how people seek information during information-intensive tasks [19]. IFT was inspired by biological theories of how animals seek food in the wild. In IFT, a *predator* (person seeking information) pursues *prey* (valuable sources of information) through a *topology* (collection of navigable paths through an information environment). What information constitutes valuable prey depends on the predator’s *information goals*. Predators find prey by following information *scent* that they infer from *cues* in the environment, such as the labels on buttons or clickable pictures that adorn navigation options. Thus, the scent of a cue is the predator’s assessment of the value and cost of information sources obtained by taking a navigation option associated with that cue.

The focus of this paper is predator *diet*, that is, the variety of information types that a predator consumes. A predator’s *information goals* define his/her “ideal” diet, but what predators actually consume depends also on what is available in the environment and how costly the information is to obtain. The relationship between cost and diet in IFT is explained well by Anderson’s notion of *rational analysis*, which is based on the idea that humans tend toward strategies that optimally adapt to the environment [1].

To help satisfy their diets, predators commonly engage in *enrichment*, that is, transforming the environment to facilitate foraging. For example, by searching on the Web, the predator enriches the environment by creating a new patch of search results, which could potentially satisfy some or all of the predator’s information goals. In addition to using search tools, other examples of enrichment include writing a to-do list on a piece of paper and running a test on a program to create a patch of relevant program output.

The earliest IFT research was in the domain of user-web interaction. For example, computational models based on IFT have successfully predicted web foraging behavior [3,6,18]. IFT has gone on to inspire practical principles and tools for designing web sites and user interfaces [24,25].

Information Foraging for Debugging Software

In the domain of software development (and especially debugging), information foraging often occurs in the context of sensemaking. The *sensemaking process* in an information-rich domain has been represented as a series of two main learning loops: foraging for information, and making

sense of the foraged information [20]. In this model, the role of IFT is central. In fact, in Grigoreanu et al.’s sensemaking study of end-user debugging [8] (which applied the Pirolli/Card sensemaking model [20]) found that the foraging loop dominated the participants’ sensemaking process.

In the software engineering community, there has been recent research focused on supporting the questions programmers ask [5,10,11,23], and these questions can be viewed as surrogates for programmers’ information goals. The software engineering analyses and tools have not been grounded in theory, but their empirical success shows that they are useful. A premise of this paper is that IFT may be able to provide a richer, more cohesive understanding of programmers’ information seeking behaviors than atheoretic efforts. Recently, we and a few others have begun investigating the efficacy of using IFT to understand programmer information-seeking (e.g., [12,13,14,15,16]). However, that work focused only on how programmers respond to cues. This paper instead investigates the whats and hows of their diets, i.e., the relationship between programmers’ information goals and debugging strategies.

METHODOLOGY

Study Data

To investigate our research questions, we analyzed a set of nine videos we collected in a previous study of professional software developers debugging in an Eclipse environment [16]. In that study, the developers used the usual Eclipse tools, plus a new IFT-based code recommender tool powered with a variety of recommendation algorithms. This setup is consistent with real-world scenarios in which developers work on unfamiliar code, such as a new team member being brought “onboard” a project, a developer on a team needing to work on code that another team member wrote, or a newcomer to an open-source project.

To summarize the study setup, each video included screen-capture video, audio of what the participant said, and video of the participant’s face. Participants “talked aloud” as they worked. Their task was to fix a real bug in the jEdit text editor, a mature open source project. None of the participants had seen the jEdit code before, and with 6468 methods, it provided a large information space in which to forage. The bug was from an actual bug report (#2548764) and regarded a problem with deleting “folded” text. Each debugging session lasted two hours with a short break half-way through. No participants completed the task, and all exhibited instances of foraging throughout the two hours.

Categorization Procedures

We used a qualitative, multi-part coding approach to analyze these videos. First, we segmented the videos into 30-second intervals, resulting in roughly 70 segments per video. (We chose 30 seconds to be long enough for participants to verbalize a goal.) We then coded each segment to identify (1) instances of foraging, (2) participants’ infor-

mation goals, and (3) participant debugging strategies, allowing multiple codes per segment. To enhance generalizability, these code sets were drawn from prior studies, as we describe below.

To ensure reliability, we followed standard inter-rater reliability practices. Two researchers first worked together on a small portion of the data to agree on coding rules. They then independently coded 20% of the segments to test the agreement level. We computed agreement using the Jaccard index, as it is suitable when multiple codes are allowed per segment, as in our case. We performed a separate coding pass (with separate reliability checks) for each code set. For each pass, agreement exceeded 80%, so the two researchers then divided up the coding of the remaining data.

Information Foraging Behavior Codes

To code whether a participant showed evidence of information foraging within a 30-second segment, we used a two-part coding process. First, we segmented around participants' utterances and coded the segments. The codes were *foraging-start*, *foraging-end*, and *foraging-ongoing*. This code set was inspired by the scent-following code set used in [14], but ours focused only on whether or not foraging occurred, and not whether scent was lost, gained, etc. We coded an utterance as *foraging-start* when participants stated an intention to pursue a particular information goal and then took accompanying action to seek that goal, like searching. We coded an utterance as *foraging-end* when participants stated that they had learned some information, or expressed giving up on a goal. We coded an utterance as *foraging-ongoing* when participants restated a previously stated goal, or said they were still looking for something.

In the second part of the coding process, we used the utterance codes from the first part to code each 30-second segment as *foraging* or *non-foraging*. A segment was *foraging* if it had an utterance coded as *foraging-start*, *foraging-ongoing*, or *foraging-end*, else it was *non-foraging*. Also, to include segments in which a participant may not have explicitly made an utterance, we also coded segments in between *foraging-start* and *foraging-end* utterances as *foraging*. However, some segments were exceptions. If a participant clearly never foraged during a segment, we coded the segment as *non-foraging*. *Non-foraging* activities included configuring Eclipse or reasoning aloud about the task. Using this coding scheme independently, two researchers achieved 82% agreement on 20% of the data before dividing up and individually coding the remaining data.

Information Goal Codes

We based the Information Goal code set on Sillito et al.'s empirically based taxonomy of 44 questions programmers ask, which Sillito et al. had grouped into four types [23]. We coded the 30-second segments against the 44 questions, and then grouped them into the four types for presentation brevity. (Results for the 44 individual questions are given in Appendix A.) Table 1 lists the types, with a few examples

of the Sillito questions that were our actual code set. We chose the Sillito questions for several reasons. First, they are a good fit for the program-debugging domain, because they categorize information needs specific to programmers. Second, they seem generalizable to a broad range of programming languages and environments, since Sillito et al. collected them from a study that covered seven different programming languages and at least eight different programming environments. Third, they are consistent with information goals identified in other studies from both programming and non-programming domains (e.g., [8,9,14,20]). Finally, they are specific and low-level, enabling a code set with the potential for high inter-rater reliability.

We coded each participant utterance in the foraging segments (as per our foraging code set above) to one of Sillito's questions. We also included a code of *other* goals, for utterances that did not match any of the questions. Using this scheme, two coders achieved 80% agreement on 20% of the data, and then split up the rest of the coding task.

The coding resulted in 384 goals coded using the Sillito question codes and 286 *other* goals. About one fourth of the utterances coded *other* were similar to one of the Sillito questions, but were not a precise match, so for reasons of rigor, we did not include them. The remaining *other* goals were about concepts (e.g., the bug's specifications, how to use the jEdit "fold" feature, the Eclipse environment, etc.) that are beyond the scope of this paper.

Information Goal Patterns

To investigate how information goals relate to each other, we categorized the information goal data into the five patterns in Table 2. Four of the patterns (*Stairstep*, *Restart*, *Pyramid*, and *Oscillate*) came from literature suggesting progressions in these sequences (e.g., [8,20,23]). The fifth pattern, *Repeat*, emerged as a common pattern during the course of our analysis.

Following the Table 2 definitions, we used a greedy pat-

Goal Type	Codes	Examples of Sillito questions
<i>1-initial:</i> Find initial focus points	Sillito questions 1–5	#2: Where in the code is the text of this error message or UI element? #5: Is there an entity named something like this in that unit?
<i>2-build:</i> Build on those points	Sillito questions 6–20	#14: Where are instances of this class created? #20: What data is being modified in this code?
<i>3-group:</i> Understand a group of related code	Sillito questions 21–33	#22: How are these types or objects related? #29: How is control getting (from here to) here?
<i>4-groups:</i> Understand groups of groups	Sillito questions 34–44	#35: What are the differences between these files or types? #43: What will be the total impact of this change?

Table 1. Information goal types with examples [23].

tern-matching algorithm (which always returned the longest possible matches) to identify instances of the patterns in the goal data. We did not allow matches that contained a gap of 5 or more minutes (i.e., 10 or more 30-second segments) between goal utterances or contained an interruption/intervention, such as the between-session break. We permitted overlapping patterns, except for instances of *Oscillate* completely contained within a *Stairstep* or *Pyramid*, and for instances of *Stairstep* completely contained within a *Pyramid*. We omitted *Oscillate* and *Stairstep* instances in these cases, because they were essential components of the containing patterns. A single author performed this analysis because the definitions were objective and the analysis automatable.

Debugging Strategy Codes

To code participant strategies, we reused Grigoreanu et al.’s debugging strategy code set [7]. We chose these strategy codes because, while being specific to the program debugging domain, each also maps cleanly to one of the three key foraging activities [19]: within-patch foraging, between-patch foraging, and enrichment. (Technically, enrichment is a between-patch foraging activity; however, in this paper, we use the term *between-patch foraging* to include only non-enrichment activities.)

Table 3 lists the strategy codes grouped by type of foraging activity. The Within-Patch strategies all involve looking for information within the contents of a single patch, such as in a Java method or web page. The Between-Patch strategies all involve navigating between different patches by selecting and clicking links, such as those provided by the recommender tool. The Enrichment strategies all involve manipulating the environment to facilitate foraging, for example, by creating a new patch of search results.

For each segment, we looked for evidence of the participant applying each strategy using indicators such as those shown in Table 3. A segment could have multiple strategy codes. Using this scheme, two coders achieved 80% agreement on 28% of the data, and then divided up the remaining data.

RESULTS

Preliminaries: How much foraging did they do?

As Table 4 shows, participants spent 50% of their 2-hour sessions foraging on average. We were unable to find prior measures of programmer foraging with which to compare this result, but Ko et al. measured time spent on *mechanics* of navigation. Their programmers spent 35% of the time on “the mechanics of navigation between code fragments” [10]. Even our participant who foraged the least still did so more than 35% of the time.

RQ1: The Whats of Programmers’ Diets

A Diversity of Dietary Whats

Although all participants had the same high-level information goal (to find the information needed to fix the bug),

their dietary preferences were diverse, as Table 5 shows. (Recall the four goal types defined in Table 1.) In aggregate, participants pursued the most goals of Type 1-initial,

Pattern	Example	Formal Definition
<i>Oscillate</i> : Back and forth between two adjacent types repeatedly.	1121212212	$O = O_1 \mid O_2$ where: $O_1 = UpDn(1,2) \mid UpDn(2,1)$ $O_2 = UpDn(2,3) \mid UpDn(3,2)$ $UpDn(a,b) = a+b+(a+b)+a^*$
<i>Stairstep</i> : From 1 up through adjacent types to at least 3.	1122223	$Stairstep = (1+2^*) + (2+3^*) + 3$
<i>Restart</i> : Jump off the Stairstep down to 1	112331	$Restart = Stairstep \ 1$
<i>Pyramid</i> : Up then down the stairsteps. Constraint: If Pyramid, then not Stairstep.	12321	$Pyramid = Pup \ Pdown \mid 2+ \ Pup \ Pdown \ 1$ where: $Pup = (1+2^*) + (2+3^*) +$ $Pdown = (3+2^*) + (2+1^*) +$
<i>Repeat</i> : One type at least 10 times.	1111111111	$Repeat = 1111111111^* \mid 2222222222^* \mid 3333333333^* \mid 4444444444^*$

Table 2. Information goal patterns. Each definition is a regular expression of Goal Type instances (+ means 1 or more instances, * means 0 or more; “|” means “or”). E.g.: 1+2+ means one or more instances of Type 1, then one or more of Type 2. We omit Type 4s next to Type 3s because 4 never followed 3 in our data.

Strategy	Example Indicators
<i>Within-Patch Strategies</i>	
Specification checking	Looking for info by reading within the bug description
Spatial	Looking for info by reading through the list of package contents in the Package Explorer
Code inspection	Looking for info by reading within a Java code file
File inspection	Looking for info by reading within a non-code file, such as a Java properties file
Seeking help-Docs	Looking for info by reading within the jEdit documentation
<i>Between-Patch Strategies</i>	
Control flow	Following control dependencies
Dataflow	Following data dependencies
Feedback following	Following method links from the recommender tool
<i>Enrichment Strategies</i>	
Code search	Creating a patch of search results with the Eclipse code search utility
Testing	Creating a patch of program output or internal state to inspect
To-do listing	Writing notes on paper
Seeking help-Search	Creating a patch of search results with an (external) web search for info on bug/code

Table 3. Debugging strategy code set [7] with example indicators for each strategy.

Participant:	P2	P3	P5	P6	P7	P8	P9	P10	P11	Mean
Time Foraging:	52%	71%	38%	63%	46%	43%	48%	42%	49%	50%

Table 4. Participants spent a large fraction of their time, ranging from 38% to 71%, foraging for information.

with slightly fewer in 2-build, and many fewer in the more complex 3-group and 4-groups. However, most participants did not conform to the aggregate: Only P6 and P9 had goal counts consistent with the aggregate.

Patterns of Dietary Relationships

Despite their dietary diversity, the progression of information goals that participants pursued often followed certain patterns (summarized in Table 6 and Figure 1; patterns defined in Table 2). Eight of the nine participants displayed one or more of the patterns, and 58% of segments in which a participant expressed a goal were part of a larger pattern. Participants exhibited a median of 1.5 patterns each, with P6 exhibiting all five.

For example, P6’s use of the Restart pattern at the end of a Stairstep is shown in the Figure 1e example. The Restart occurred when his Stairstep progression culminated in gaining the information he sought about the handleMessage method’s relationship to the editor (a Type 3-group goal):

Goal Type	P2	P3	P5	P6	P7	P8	P9	P10	P11	Total
1-initial	6	76	0	34	18	8	18	8	2	170
2-build	3	1	2	24	34	17	16	15	11	123
3-group	2	2	2	3	2	3	15	9	11	49
4-groups	13	1	0	0	0	9	0	3	16	42
Total	24	80	4	61	54	37	49	35	40	384

Table 5. Number of segments spent on the (codeable) types of information goals. Gray highlights each participant’s most-pursued goal type.

Pattern	P2	P3	P5	P6	P7	P8	P9	P10	P11
Repeat	1(4)	2(1)		1(1)	1(2)			1(2)	1(4)
Oscillate				1(1,2)	2(1,2)			1(3,2)	1(3,2)
Stairstep				1					
Pyramid				1		1	2	1	
Restart				1					

Table 6: Frequency of pattern instances exhibited by each participant. The numbers in parentheses indicate the type of goals within the pattern (e.g., 1(3,2) in the Oscillate row indicates patterns like 33322322, as defined in Table 2).

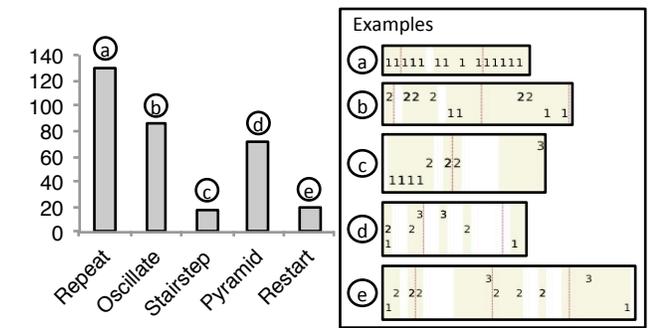


Figure 1. Frequency of goal patterns. Y-axis is count of segments in each pattern. Each bar is labeled with an example from the participants’ videos. The beige background denotes foraging; white is non-foraging (e.g., studying the code that has been found); and numbers denote the goal types.

P6: “So this (handleMessage) is handling some events for the editor.”

This was what P6 had wanted to know, so he then changed to a new line of foraging, thus dropping down to a Type 1-initial goal:

P6: “But I don’t know how the menu is hooked up to this. ... I wonder if there is some method that might be named ‘delete lines’ ...” [P6 starts searching in package explorer.]

Some of these patterns were predicted by the literature. Sillito et al. [23] suggested one progression: find an initial focus (1-initial), then build on it (2-build), then understand a group of related foci (3-group), and finally understand groups of groups (4-groups). Other empirical studies have found a similar progression from 1-initial to 2-build, including our previous work on information foraging during debugging (characterized there as “debugging modes”) [14], and earlier work on how people seek information in web environments (summarized in [9]). Furthermore, the notion of progressing from Type 1-initial to 2-build to 3-group to 4-groups is consistent with prior results from applying Pirolli and Card’s sensemaking model [20] to intelligence analysts and to end-user debuggers [8].

However, participants did not usually organize their foraging in the ways suggested by the above literature: Stairstep, Pyramid, and Restart together accounted for only 22% of the pattern segments. In fact, only four of the participants used any of them at all! This finding suggests that idealized progressions outlined in prior research miss much of how programmers forage for information in code, at least in the widely used Eclipse environment.

In contrast to the patterns from the literature, the Repeat pattern, which emerged from our study, occurred frequently. In Repeat, a participant spent extended periods following one information goal type. 6 of the 9 participants exhibited this pattern—greater usage than any other pattern.

Why did participants exhibit the above patterns? To answer this question, we need two pieces of information: what strategies they used for their foraging, and how those strategies came together with their goals and goal patterns. We discuss each of these in turn in the next two sections.

RQ2: The Hows: Strategies during Foraging

Recall from Methodology (Table 3) that each debugging strategy maps to an IFT activity: within-patch foraging, between-patch foraging, and enrichment. Table 7 shows each participant’s strategy usage by IFT category.

Debugging Strategies Meet IFT

Since much of the prior IFT research has focused on between-patch scent following (e.g., [3,13]), we were surprised that only 24% of participants’ foraging fell into that category. Participants spent considerably more time foraging within patches and performing enrichment.

As Table 7 shows, participants used a diverse mix of strate-

gies (median of 8 different strategies); however, each foraging category had clearly dominant strategies. Spatial was the participants' primary Within-Patch strategy; Control Flow was their primary Between-Patch strategy; and Code Search and Testing were together (but especially Testing) their primary Enrichment strategies.

What Participants Used Enrichment For

Enrichment is an activity wherein the predator changes the environment to facilitate foraging [19]. The participants changed their environments in two ways. Code Search, Seek Help-Search, and To-Do Listing involved creating a patch of *links* to other patches for the predator to *navigate*. In contrast, Testing involved creating patches of information *content* for the predator to *process*.

Most participants strongly favored one or the other of these types of enrichment strategies. In particular, they either favored creating patches of linked search results with Code Search, or creating patches of runtime state information with Testing. In fact, over half of the participants used only one of Code Search or Testing. For example, Participant P7 used Code Search repeatedly, trying to find methods that implemented line deletion and folding in jEdit:

P7: "Let's see if I can find something like what is in that bug report." [Searches for *delete lines*. No results.] "Let's just look for 'explicit fold'." [Searches for *explicit fold*.] "Finally, something that actually has to do with folding..."

In contrast, P5 stepped through program runs repeatedly, collecting information about its internal state:

P5: [Looks at the debugger's Variable Watch view.] "*lineCount* is zero." [Reads code.] "I'm going to step into that

Strategy	P2	P3	P5	P6	P7	P8	P9	P10	P11	Total
<i>Within-Patch Strategies</i>										
Spec. Checking	2	9	0	11	0	0	0	3	0	25
Spatial	25	39	5	28	31	14	47	19	12	220
Code Inspection	4	9	10	16	17	15	22	7	30	130
File Inspection	0	6	0	4	0	0	0	3	0	13
Seek Help-Doc	4	0	0	0	0	0	0	2	0	6
Total:	35	63	15	59	48	29	69	34	42	394
<i>Between-Patch Strategies</i>										
Control Flow	19	1	18	14	20	27	23	14	21	157
Data Flow	0	0	5	1	2	4	0	7	5	24
Feedback Follow.	4	8	12	5	6	4	6	1	6	52
Total:	23	9	35	20	28	35	29	22	31	232
<i>Enrichment Strategies</i>										
Code Search	0	51	0	29	33	4	0	12	0	129
Testing	36	0	34	14	5	37	30	22	45	223
Todo Listing	1	1	0	1	1	2	0	0	5	11
Seek Help-Search	0	4	0	0	0	0	0	0	0	4
Total:	37	56	34	44	39	43	30	34	49	366
Overall Total:	95	128	84	123	115	107	128	90	122	992

Table 7. Usage (segment counts) of each strategy during foraging. Gray cells indicate the maximum frequency by participant and by strategy category. Although participants foraged in a total of 660 segments, the overall total of strategy segments (992) is greater because participants used multiple strategies during some segments.

(method)" [Steps.] "*count* is greater than—now *count* is zero. [Steps again.] "I'm stepping through the code. ... I'm trying to understand what this code is doing."

Despite prior findings about users' preference for searching (e.g., [2]), four of the nine participants used neither Code Search nor Seek Help-Search. This lack of searching cannot be because the task was too easy (no one finished) or the code base was too small (it had 6468 methods). However, earlier findings on web information processing [9] may explain this result. Hearst points out that, in many cases, browsing works better than searching because it is mentally less costly to *recognize* a piece of information than it is to *recall* it, and recall is often needed to formulate an effective search query. Consistent with Hearst's observation, every participant used the Code Inspection strategy.

Go-To Strategies for Foraging

Reconsidering Table 7 from a most-used perspective, some strategies stand out as having been used particularly often for one or more aspects of foraging. The leftmost four (white) columns of Table 8 summarize.

RQ3: Whats Meet Hows: Dietary Strategies

Strategies by Goal Type

Table 9 and Figure 2 tie all 12 of the strategies back to the participants' dietary goals. As the table and figure show, some strategies were strongly tied to particular goal types. For example, Specification Checking was used only for Type 1-initial goals, and Code Inspection was used primarily for Type 2-build goals. Figure 2 shows that participants used Code Search (labeled a) and Spatial (labeled b) more than the other strategies with their Type 1-initial goals. From a patch perspective, Spatial seemed particularly suited to helping participants cope with large patches, and Code Search with large spaces of patches. For example, P6 spent considerable time performing Spatial in the Package

Strategy	How many used it?	Top strategy for...			
		... which participants	... which IFT category	... which Goal Type	... which Patterns
<i>Within-Patch Strategies</i>					
Spatial	all 9	P9	Within	2-Build	Pyramid
Code Inspect.	all 9	-	-	-	-
<i>Between-Patch Strategies</i>					
Control Flow	all 9	-	Between	-	Restart
Feedback Follow.	all 9	-	-	-	-
<i>Enrichment Strategies</i>					
Code Search	5	P3, P6, P7	-	1-initial	Repeat, Oscillate, Stairstep
Testing	8	P2, P5, P8, P10, P11	Enrich.	3-group, 4-groups	-

Table 8. These 6 strategies (out of 12) stood out. Each of these was used by everyone, was at least one person's most-used strategy, or was the top strategy for an IFT category.

Explorer view (a patch containing hundreds of lines), looking for a Java class on which to focus:

P6: “I keep thinking this menu package gotta be involved somehow.” [P6 scans down the list of Java classes inside the menu package in Eclipse’s Package Explorer view.]

P3, on the other hand, applied Code Search to search the

Strategy	Information Goal Type				Total
	1-initial	2-build	3-group	4-groups	
<i>Within-Patch Strategies</i>					
Spec Checking	24	0	0	0	24
Spatial	92	62	23	8	185
Code Inspection	13	56	20	4	93
File Inspection	10	0	0	0	10
Seeking Help-Docs	2	0	0	0	2
Total:	141	118	43	12	314
<i>Between-Patch Strategies</i>					
Control Flow	21	46	14	9	90
Dataflow	1	4	3	1	9
Feedback Following	13	15	7	1	36
Total:	35	65	24	11	135
<i>Enrichment Strategies</i>					
Code Search	104	47	2	1	154
Testing	25	26	26	30	107
To-Do Listing	1	4	1	3	9
Seeking Help-Search	0	0	0	0	0
Total:	130	77	29	34	270
Overall Total:	306	260	96	57	719

Table 9. Strategy usage by goal types. Gray highlights the maximum strategy usage for each goal type. The overall total (719) is greater than the total foraging segments (660) because some segments contained multiple strategies. The total for Seeking Help-Search was 0 because none of the strategy’s 4 instances co-occurred with a goal statement.

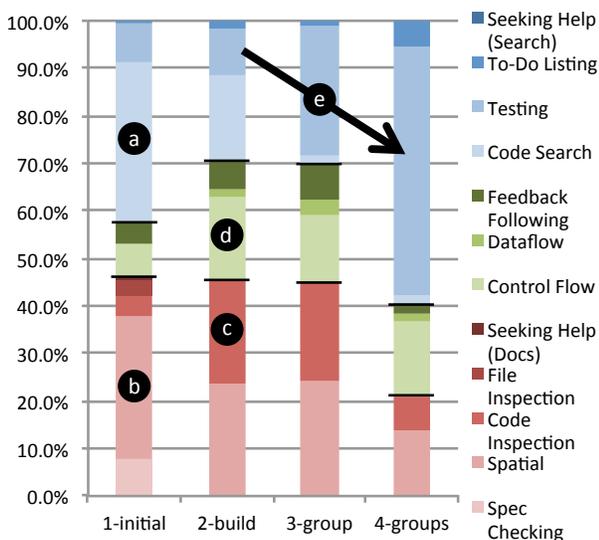


Figure 2: Strategy proportions by goal type. Strategies are color-coded, with black bars separating the IFT categories. Red: Within-Patch. Green: Between-Patch. Blue: Enrichment.

6468 methods for code related to deleting lines in jEdit:

P3: “I would imagine that I would look for the word ‘delete’ perhaps, especially given that that’s the term that’s used in the menu.” [Executes a search for *delete*.]

Participants tended toward different strategies for the Type 2–4 goals, which express progressively deeper relationships among code entities. For example, Figure 2 shows the shift away from Code Search and Spatial, and toward Code Inspection (c) and Control Flow (d) for Type 2-build and Type 3-group goals. Testing in particular (e) increased markedly from Type 2-build to Type 4-groups goals.

Considering participants’ goal patterns in the context of their strategies (summarized in Table 10) sheds additional light on why the patterns emerged.

Pattern Repeat: Constant Goal Type, Constant Strategies

Pattern Repeat, repeated pursuit of a single goal type, was also characterized by repeated participant use of a constant handful of *strategies*. The Repeat instances occurred in two cases. In one case, participants’ debugging strategies were producing the desired goals efficiently, i.e., at such low cost to the participants that staying with that goal type and strategy was a good way to optimize their costs. In the other case, their strategy for that goal type was so ineffective, they needed a long time to fulfill that type of dietary need.

As an example of the first case, P7 followed the Repeat pattern on Type 2-build goals using three strategies continuously: Spatial, Code Inspection, and Control Flow. Eclipse supports all three with low-cost navigation tools, such as

Pattern	Participant	Strategy		
		Within-Patch	Between-Patch	Enrichment
Repeat(1)	P3	70%	6%	71%
	P6	75%	0%	75%
Repeat(2)	P7	83%	72%	48%
	P10	50%	38%	63%
Repeat(4)	P2	40%	10%	80%
	P11	30%	40%	90%
	Median:	60%	24%	73%
Oscillate(1,2)	P6	79%	8%	63%
	P7	74%	57%	60%
Oscillate(3,2)	P10	40%	47%	73%
	P11	100%	48%	62%
	Median:	77%	47%	62%
Pyramid	P6	62%	57%	76%
	P8	53%	53%	88%
	P9	100%	21%	7%
	P10	53%	29%	76%
Median:	57%	41%	76%	
Restart	P6	61%	57%	70%
Stairstep	P6	62%	57%	76%

Table 10. Percentage of goal-pattern segments that co-occurred with each category of strategy. Recall that multiple strategies were allowed per segment. Gray denotes the maximum category for each pattern.

one-click navigation to the declaration of any class, method, or variable. P7 used these features to efficiently fulfill his Type 2-build goals, and fulfilled multiple goals, often building from one goal to the next using the same strategies.

When participants followed the Repeat pattern on goals of Type 1-initial or of Type 4-groups, their strategies were still constant, but not as fruitful. In the cases involving Type 1-initial, participants used Code Search (Enrichment) and Spatial (Within-Patch) extensively, but not particularly fruitfully, looking for a place to start. For example, Figure 3 shows P3 repeatedly using Code Search to find an initial starting point. Likewise, in P11's use of Repeat on his Type 4-groups goals, he used Testing across numerous segments of the pattern, trying to understand the relationship between changes he had made and the rest of jEdit's functionality. He pieced the information together by laboriously gathering it in small bits, one execution of the program at a time.

Pattern Oscillate: Changing Strategies to Dig Deeper

For the participants who followed the Oscillate pattern on Type 1-initial and 2-build goals, the story was similar to Repeat on Type 1-initial, except the oscillators tended to seek additional information from their search results. In particular, the oscillating participants would typically do a code search, explore the results a bit, decide they were on the wrong track, and return to searching. Unlike the Repeat pattern, the participants we observed within the Oscillate pattern switched strategies rapidly along with their goals. Figure 4 illustrates this behavior for P6.

Patterns for Enrichment and Goal Switching

Table 10 suggests that Enrichment tended to drive the inter-related Pyramid (up then down the stairs), Restart (stairs followed by starting again), and Stairstep (climb the stairs) patterns. Participants following the Pyramid pattern used the Enrichment strategies of Code Search and Testing equally often, but P6's instances of Stairstep and Repeat were characterized by almost exclusive use of the Code Search strategy. (Only P6 followed these two patterns.)

All three patterns were characterized by rapid goal fulfillment followed by a rapid switch to the next goal. This rapid fulfillment and initiation of the next goal type is consistent with our previous findings pointing to the reactivity of foraging in this domain [13,16].

The Most-Used Strategies' Strengths

This brings us to the particular strengths of different strategies. Refer back to Table 8; the rightmost (shaded) columns include the goal types and patterns we have just discussed for the most-used foraging strategies. As the table shows, certain classic debugging strategies were used heavily in *foraging* but often were concentrated into dietary niches. For example, Code Inspection and Feedback Following were generalists—used by everyone, but not the top in any particular IFT category, any goal type, or any pattern. In contrast, Code Search was a specialist, dominating some of

the patterns and one of the goal types, but still used by only half the participants.

DISCUSSION

Generalizability

As in any empirical study, our results may have been influenced by the environment the participants used, the tools available to them, the task they worked on, etc. Issues like

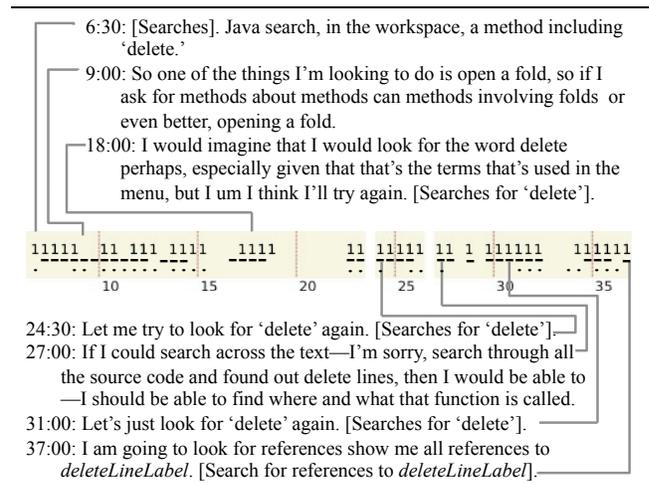


Figure 3. P3 continuously used Code Search (underlined) to find code relevant to deleting lines of text. He often complemented it with Spatial (dots). The beige background denotes foraging; white is non-foraging and the numbers indicate the minutes in the session.

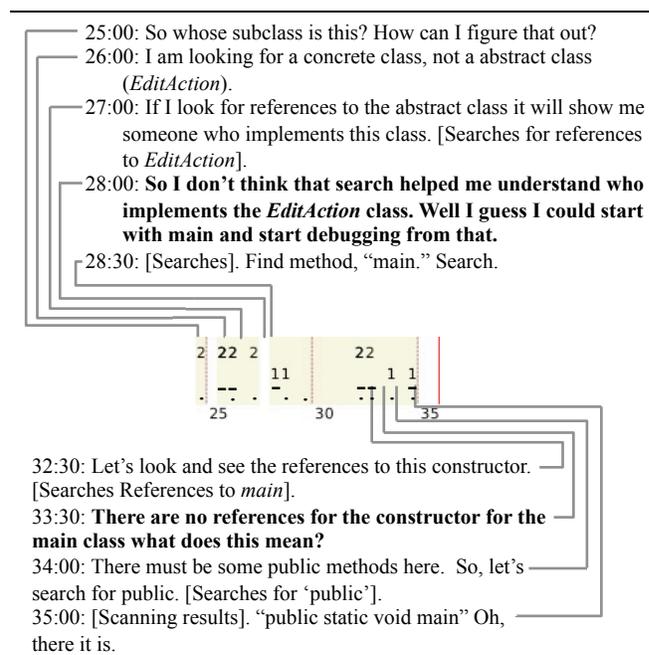


Figure 4. The Oscillate pattern for P6. The abandonment of goals is highlighted in bold. The underlines are segments with Code Search. The dots are segments with Spatial. Note that strategies alternate with the goal types.

these can be resolved only through additional studies. However, our methodology was designed to strengthen generalizability through the use of code sets and methodological conventions from other pertinent studies (e.g., [7,8,22]), and through the use of realistic elements: The software project was a real open source project; the bug was from a real bug report; the participants were experienced professionals (not students) using a popular IDE; and the participants worked alone on fairly long tasks (similar to what they would encounter professionally, even in a team context).

Diet Whats: The Long Tail

Participants' dietary needs varied greatly. This variety was not only between participants, but also within each participant's session from one moment to the next.

Our participants' diverse diets are reminiscent of the highly varied and personal diets reported by the Evans/Card study [4]. Evans and Card attributed this finding to a "long tail" demand curve, in which an abundance of available information makes it possible for people to satisfy their own individual, even quirky, desires for information. However, in the Evans/Card study, people foraged as part of their own individual tasks. Interestingly, we saw the same phenomenon with our participants, even though they all had the *same* overall goal (to fix the bug).

The participants' sometimes stubborn pursuit of particular information goals—tolerating very high costs even when their efforts showed only meager promise of delivering the needed dietary goal—highlights an important difference in the software domain versus other foraging domains: Programmers' dietary needs are often very specific. For an information forager on the Web, one dictionary page is often as good as another. But for a programmer trying to fix a bug, only very particular information about very specific code locations will help them in their task. This high dietary selectiveness in this domain may explain the high costs programmers were sometimes willing to pay.

Whats Meet Hows: Diet-Specific Strategies

Our results identified particular strategies that participants preferred for certain information goals. Of the 12 strategies we coded, 6 dominated, but in different ways.

Among the Within-Patch strategies, two strategies, Spatial (scanning lists) and Code Inspection (reading code), showed distinct associations to particular goal types. Spatial was ubiquitous across all goal types—it seemed that there was almost always some patch of information that a participant could scan. In contrast, Code inspection was particularly tied to Type 2-build and 3-group goals. Apparently, participants tended not to read code in detail when looking for an initial place to start (1-initial) or when trying to understand more complex relationships among groups of entities (4-groups). Instead, they dug into the code only when they needed information about more basic relationships (2-build and 3-group).

Turning to the Between-Patch strategies, participants applied Control Flow (following control dependencies) for all goals that involved understanding relationships between code entities (i.e., goal types 2–4). The proportion of participants who used the strategy relative to other strategies held steady, whether they were building up a basic understanding about a code entity (2-build) or understanding complex inter-relationships among groups of entities (4-groups).

Participants used the two most frequently used Enrichment strategies, Code Search (via a search utility) and Testing (running code), for different purposes. They used Code Search heavily for finding initial starting places and building upon them (Types 1 and 2). In contrast, they favored Testing for acquiring more complex information about the relationships between entities and between groups of entities (Types 3 and 4).

Participants' goal patterns reveal a close relationship between these Enrichment strategies and many of the goal patterns in Table 2. For example, participants who followed Repeat on Type 1-initial goals and who followed Oscillate on Type 1-initial and 2-build goals were generally using Code Search (Enrichment) repetitively (expending much effort with little success), looking for code relevant to the bug to investigate in more depth. Similarly, participants who used Repeat on Type 4-groups goals were generally Testing (Enrichment) by repetitively stepping through executions of the program over and over to build up information about the program's internal execution state. Overall, Enrichment strategies were heavily used in all patterns.

CONCLUSION

In this paper, we considered *what* programmers want in their diets and *how* they forage to fulfill each of their dietary needs. Some results this diet perspective revealed were:

RQ1 (whats):

- *Diversity*: Even though all participants were pursuing the same overall goal (the bug), they sought highly diverse diets. This suggests a need for debugging tools to support "long tail" demand curves of programmer information.
- *Dietary patterns*: Most foraging fell into distinct dietary patterns—including 78% in a new pattern not previously proposed in the literature.

RQ2 (hows):

- *Foraging strategies*: Participants spent only 24% of their time following between-patch foraging strategies, but between-patch foraging has received most of the research attention. This suggests a need for more research on how to support within-patch and enrichment foraging.
- *Search unpopularity*: Search was not a very popular strategy, accounting for less than 15% of participants' information foraging—and not used at all by 4 of our 9 participants—suggesting that tool support is still critical for non-search strategies in debugging.

RQ3 (what meets how):

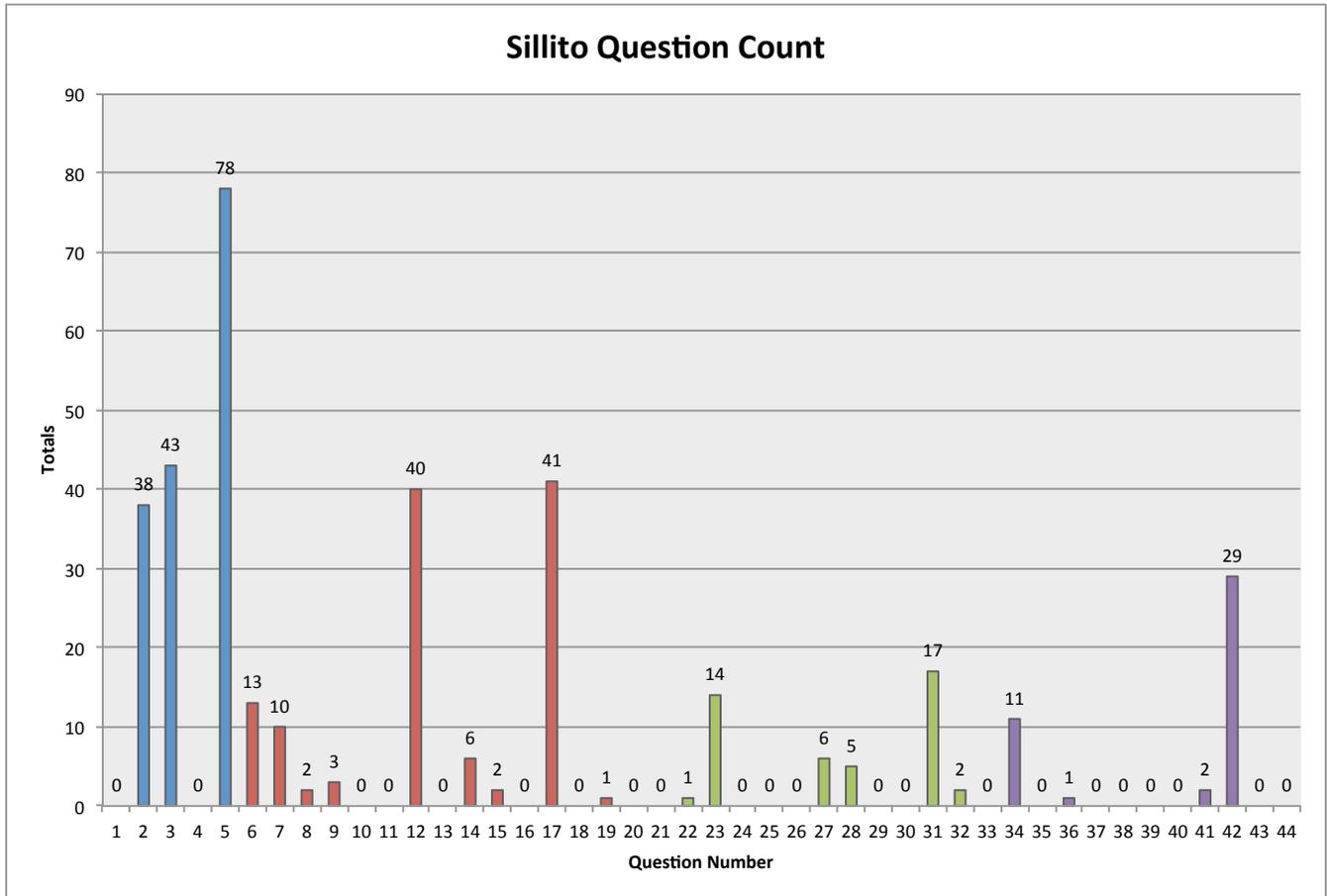
- *Strategies' diet-specificity*: Some foraging strategies were of general use across information goal types, but others were concentrated around particular dietary niches. This suggests tool opportunities; for example, tools aimed at supporting a particular strategy may be able to improve performance by focusing on the strategy's dietary niche.
- *Cost of selectivity*: Participants stubbornly pursued particular information in the face of high costs and meager returns. This emphasizes a key difference between software development and other foraging domains: the highly selective nature of programmers' dietary needs.

As Evans and Card summarize from Simon: "For an information system to be useful, it must reduce the *net* demand on its users' attention" [4]. Our results suggest that the diet perspective can help reveal when programming tools help to reduce this net demand—and when they do not—during the 50% of debugging time programmers spend foraging.

REFERENCES

1. Anderson, J. *The Adaptive Character of Thought*. Lawrence Erlbaum Associates, 1990.
2. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S. Two studies of opportunistic programming: Interleaving web foraging, learning and writing code. *Proc. CHI*, ACM (2009), 1589-1598.
3. Chi, E., Pirolli, P., Chen, K., and Pitkow, J. Using information scent to model user information needs and actions on the web. *Proc. CHI*, ACM (2001), 490-497.
4. Evans, B. and Card, S. Augmented information assimilation: Social and algorithmic web aids for the information long tail. *Proc. CHI*, ACM (2008), 989-998.
5. Fritz, T. and Murphy, G. Using information fragments to answer the questions developers ask. *Proc. ICSE*. ACM/IEEE (2010), 175-184.
6. Fu, W.-T. and Pirolli, P. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Human-Computer Interaction* 22, 4 (2007). 355-412.
7. Grigoreanu, V., Burnett, M. and Robertson, G. A strategy-centric approach to the design of end-user debugging tools. *Proc CHI*, ACM (2010), 713-722.
8. Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., Kwan, I. End-user debugging strategies: A sensemaking perspective, *ACM Trans. Comp.-Human Interaction* 19, 1, Article 5, (2012), 28 pages.
9. Hearst, M. User interfaces for search, In *Modern Information Retrieval*, 2nd Edition, ACM Press, (2011).
10. Ko, A., Myers B., Coblenz, M., Aung, H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Soft. Eng.* 33, (2006), 971-987.
11. LaToza, T. and Myers, B. Visualizing call graphs. *Proc. VL/HCC*, IEEE (2011), 117-124.
12. Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks, *Proc. CHI*, ACM (2008), 1323-1332.
13. Lawrance, J., Burnett, M., Bellamy, R., Bogart, C. and Swart, C. Reactive information foraging for evolving goals. *Proc. CHI*, ACM (2010), 25-34.
14. Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., Fleming, S. How programmers debug, revisited: An information foraging theory perspective, *IEEE Trans. Soft. Eng.* (2013), [DOI: 10.1109/TSE.2010.111].
15. Niu N., Mahmoud, A. and Bradshaw, G. Information foraging as a foundation for code navigation. *Proc. ICSE*, ACM/IEEE (2011), 816-819.
16. Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R. and Swart, C. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. *Proc. CHI*, ACM (2012), 1471-1480.
17. Piorkowski, D., Fleming, S., Kwan, I., Burnett, M., Scaffidi, C., Bellamy, R., Jordhal J. The whats and hows of programmers' foraging diets, *Proc. CHI*, ACM (2013).
18. Pirolli, P. Computational models of information scent-following in a very large browsable text collection. *Proc. CHI*, ACM (1997), 3-10.
19. Pirolli, P. and Card, S. Information foraging. *Psychological Review* 106, (1999), 643-675.
20. Pirolli, P. and Card, S. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. *Proc. Int'l. Conf. Intelligence Analysis*. MITRE Corp. (2005).
21. Pirolli, P. and Fu, W-T. SNIF-ACT: A model of information foraging on the world wide web. *Proc. User Modeling*, LNCS. Springer-Verlag Berlin. (2003).
22. Romero, P., du Boulay, B., Cox, R., Lutz, R., and Bryant, S. Debugging strategies and tactics in a multi-representation software environment. *Int'l J. Hum.-Comp. Studies* 65, Academic Press, (2007).
23. Sillito, J., Murphy, G. and De Volder, K. Questions programmers ask during software evolution tasks. *Proc. FSE*, ACM (2006), 23-34.
24. Spool, J., Profetti, C. and Britain, D. Designing for the scent of information, *User Interface Eng.*, (2004).
25. Teo, L., John, B., Blackmon, M. CogTool-Explorer: A model of goal-directed user exploration that considers information layout, *Proc. CHI*, ACM (2012).

APPENDIX A



The numbers on the bottom of the figure refer to the following 44 questions from Sillito et al. [23].

1. Which type represents this domain concept or this UI element or action?
2. Where in the code is the text in this error message or UI element?
3. Where is there any code involved in the implementation of this behavior?
4. Is there a precedent or exemplar for this?
5. Is there an entity named something like this in that unit (for example in a project, package or class)?
6. What are the parts of this type?
7. Which types is this type a part of?
8. Where does this type fit in the type hierarchy?
9. Does this type have any siblings in the type hierarchy?
10. Where is this field declared in the type hierarchy?
11. Who implements this interface or these abstract methods?
12. Where is this method called or type referenced?
13. When during the execution is this method called?
14. Where are instances of this class created?
15. Where is this variable or data structure being accessed?
16. What data can we access from this object?
17. What does the declaration or definition of this look like?
18. What are the arguments to this function?
19. What are the values of these arguments at runtime?
20. What data is being modified in this code?
21. How are instances of these types created and assembled?
22. How are these types or objects related? (whole-part)
23. How is this feature or concern (object ownership, UI control, etc) implemented?
24. What in this structure distinguishes these cases?

25. What is the behavior these types provide together and how is it distributed over the types?
26. What is the “correct” way to use or access this data structure?
27. How does this data structure look at runtime?
28. How can data be passed to (or accessed at) this point in the code?
29. How is control getting (from here to) here?
30. Why isn’t control reaching this point in the code?
31. Which execution path is being taken in this case?
32. Under what circumstances is this method called or exception thrown?
33. What parts of this data structure are accessed in this code?
34. How does the system behavior vary over these types or cases?
35. What are the differences between these files or types?
36. What is the difference between these similar parts of the code (e.g., between sets of methods)?
37. What is the mapping between these UI types and these model types?
38. Where should this branch be inserted or how should this case be handled?
39. Where in the UI should this functionality be added?
40. To move this feature into this code what else needs to be moved?
41. How can we know this object has been created and initialized correctly?
42. What will be (or has been) the direct impact of this change?
43. What will be the total impact of this change?
44. Will this completely solve the problem or provide the enhancement?