To appear: IEEE International Conference on Software Maintenance and Evolution,
Bremen, Germany, Sep 29 - Oct 1, 2015.

To Fix or to Learn? How Production Bias Affects Developers' Information Foraging during Debugging

David Piorkowski[1], Scott D. Fleming[2], Christopher Scaffidi[1], Margaret Burnett[1],
Irwin Kwan[1], Austin Z. Henley[2], Jamie Macbeth[3], Charles Hill[1], Amber Horvath[1]

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

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

[3]Clemson University
Clemson, SC, USA
jmacbet
@clemson.edu

***Abstract*— Developers performing maintenance activities must balance their efforts to learn the code vs. their efforts to actually change it. This balancing act is consistent with the "production bias" that, according to Carroll's minimalist learning theory, generally affects software users during everyday tasks. This suggests that developers' focus on efficiency should have marked effects on how they forage for the information they think they need to fix bugs. To investigate how developers balance fixing versus learning during debugging, we conducted the first empirical investigation of the interplay between production bias and information foraging. Our theory-based study involved 11 participants: half tasked with fixing a bug, and half tasked with learning enough to help someone else fix it. Despite the subtlety of difference between their tasks, participants foraged remarkably differently—making foraging decisions from different types of "patches," with different types of information, and succeeding with different foraging tactics.**

Keywords- debugging; information foraging; theory meets tools

I. Introduction

Software maintenance invariably requires some learning of the current code before making changes, so as to plan correct and efficient changes. For example, one study suggested that methodical exploration of code prior to making maintenance changes can cut the time of actually coding in half [34]. Because of this connection between comprehension and maintenance, a wide range of tools aim at aiding program exploration and comprehension during maintenance and evolution (e.g., [2][11][12][15][21][25][42][43][44]).

It is therefore troubling that a recent observational field study of work practices revealed that "wherever possible, developers seem to prefer strategies that avoid comprehension" of existing code [35]. Specifically, this study found that developers frequently tried to move forward with coding as quickly as possible (what could be considered a form of satisficing [37]), with a minimal amount of activity invested ahead of time in exploring the code they were about to modify. Such results are consistent with those of other studies [3][17][23].

Based on such findings, Maalej et al. concluded:

> Software comprehension is a hard and time-consuming task and consequently is avoided whenever possible. This indicates that Carroll's minimalist theory [Carroll 1998], which suggests people put in the minimum effort to maximize their outcome, is applicable … We think that researchers should consider developers as users and investigate how 'user-developers' analyze application behavior, how they relate observations to code, and how this behavior can be supported by tools [23].

This tendency of software developers to view learning as a costly task detracting from their efficiency is called *production bias* in Carroll's theory [5][6]. To date, the effects of production bias on developers have not been investigated in detail.

In this paper, we investigate these effects and their implications for tool design. Our investigation is grounded not only in Carroll's theory, but also in Information Foraging Theory (IFT) [29]. IFT provides a conceptual framework describing how people in an information environment, such as an IDE, seek information. For developers faced with a bug, the information they seek can include how to reproduce the bug, what causes the bug, where to fix the bug, and whether similar bugs were fixed elsewhere [23].

We performed our investigation by conducting a qualitative laboratory study. One group of developers was tasked with fixing a bug, whereas the other group was tasked with learning enough about the bug to help someone else fix it. We assigned people these differing tasks to reveal and analyze differences in their behaviors from both an IFT and a production bias perspective to address four research questions:

- *RQ1 (information goals)*: How does trying to fix a bug versus trying to learn about the bug affect the *types of information* that developers seek?
- *RQ2 (information patches)*: How does trying to fix a bug versus trying to learn about the bug affect *where in the environment* developers make foraging decisions?
- *RQ3 (information cues)*: How does trying to fix a bug versus trying to learn about the bug affect the *types of cues* developers attend to when making foraging decisions?
- *RQ4 (foraging tactics)*: How does trying to fix a bug versus trying to learn about the bug affect the *tactics* that developers use in making their foraging decisions?

II. Background

Viewing developers' code exploration during debugging through the lens of a theory can provide a conceptual frame-

work for thinking about what aspects of their behavior might vary depending on the presence of production bias. In essence, IFT [29] gives a meaningful way to “carve up” the information environment and the developers’ actions within that environment, then to investigate differences among developers in terms of how they use the environment. IFT is an appropriate theory for this purpose because it has already proven useful for explaining and predicting how developers find information during maintenance, in ways beneficial to tool design (e.g., [8][9][10][18][19][20][28]).

IFT’s main constructs consist of a *predator* (a human forager) who seeks *prey* (information) within an information environment made up of information *patches* (such as documents and screens of information, as in Fig. 1) connected by *links*. The predator navigates among these patches by traversing the links, which involves taking some navigation action, such as clicking or scrolling. Each link has a *cost* of traversal (the time to get from one patch to the other) that is influenced by both system performance and the human’s cognitive and physical speed.

The within-patch constructs consist mainly of *information features* (e.g., words, phrases, and graphics), 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* associated with outgoing links to other patches. Cues (e.g., 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), leading to a high density of cues in such environments.

These constructs are tied together via IFT’s central proposition about how the predator forages for information.

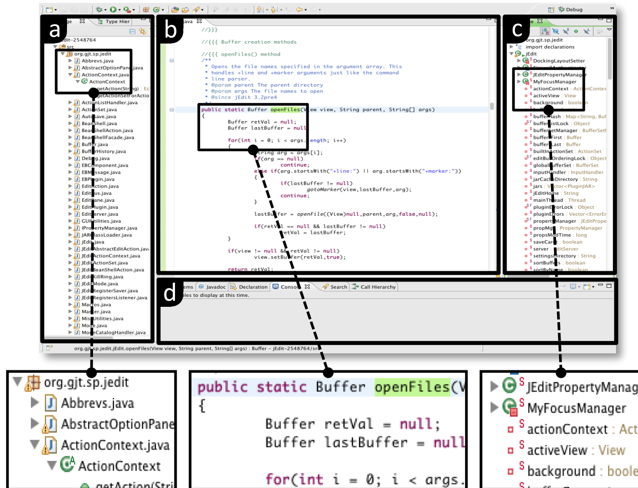

Fig. 1. An information environment (Eclipse) as a developer (predator) might see it during debugging. Patches of information content are visible in the (a) Package Explorer, (b) Editor, and (c) Outline View (plus a region (d) where other patches can appear). As the blow-ups of (a) shows, each item in 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.

The predator’s available choices at a given moment are (1) to process information features within the current patch, (2) to navigate to another patch via a link, or (3) to enrich the environment, such as by decorating a patch (e.g., by annotating it), or by creating a new patch (e.g., by querying a search engine to create a list of search results). IFT’s central proposition is that the predator will tend to choose the action that maximizes the expected value of information gained per expected cost of interaction.

However, the predator is not omniscient and may not know the actual value and cost that a choice will yield. Thus, the predator makes choices that follow from his/her *expectations* about the value and cost. The predator may base such choices, for example, on whatever he/she infers from the available cues.

The predator’s inferential judgment of value and cost are captured by IFT’s final construct: *information scent*—that is, the predator’s assessment of the costs vs. benefits (i.e., value) that following a link will yield, given the cues associated with that link. In the second choice above, navigating to another patch via a link, the predator uses scent to decide which link to follow. Specifically, IFT states that the predator will choose the link with the strongest scent (greatest benefit per cost).

Although there have been a number of promising research results that leverage IFT as the basis for models and tools (e.g., [7][20][27][28][31]), to date, IFT has not specified how people will forage, or how tools should help people forage, in the face of production bias. The theory suggests only that foraging behavior would change *if* production bias affects how people perceive the value and cost of patches and cues (i.e., scent). Our paper fills that gap by uncovering both *whether* and *how* production bias’s tension of learning vs. efficiency affects information foraging during software maintenance.

III. Methodology

To model Minimalist Learning Theory’s tension between learning vs. “doing,” we randomly assigned each participant to one of two treatments: Fix or Learn. We told participants in the Fix treatment group to fix a particular bug in a program. We told participants in the Learn treatment group to learn enough information about that same bug to be able to on-board a programmer new to the team—that is, enough to “help the new programmer fix the bug.” Thus, both groups needed to find the same information, but only the Fix group was asked to actually fix the bug. Thus, for the Learn group, we framed learning “enough” as an end in itself.

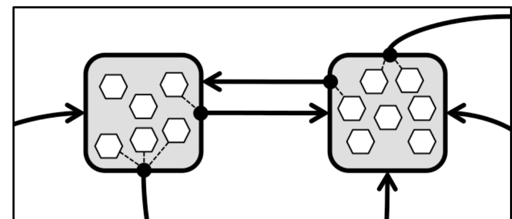

Fig. 2. Conceptual depiction of an information environment with two information patches (rounded boxes) interconnected by links (directed edges). Each patch contains information features (hexagons), with some that are also cues (connected to outgoing links).

However, we did not control how much fixing or learning participants actually did. Indeed, both treatment groups would need to do some learning/comprehension of the buggy code, and would have to decide when they had learned "enough." Also, we did not stop any participants from fixing: when Learn participants asked if they could fix the bug, we told them to feel free to do whatever they felt was necessary to learn what they needed to learn.

A. *Procedure*

The code that both groups worked with was from the jEdit project, an open source code editor consisting of 98,652 non-comment lines of Java code. All participants received a copy of bug report #3223 from jEdit's issue-tracking system, which described a problem with deleting "folded" code. All participants had access to the same tools, which consisted of the Eclipse integrated development environment and other software commonly found on a Windows PC, including a web browser with unrestricted access to the Internet.

Randomly dividing the 11 participants into two treatments resulted in 6 Fix participants and 5 Learn participants. Each participant had an individual session, lasting at most 2 hours. Throughout the session, we collected video recordings of the participant as well as screen-capture video. First, the participant filled out a background questionnaire, and we briefly explained what they should do (to learn "enough" or to fix, depending on treatment). Next, the participant worked for 30 minutes while "talking aloud."

Participants' foraging decisions were the moments of trade-off (e.g., forage to fix, to learn this, to learn that, etc.). Thus, following a short break we conducted a semi-structured retrospective interview with each participant by playing back all of the screen-capture video, then asking why the participant made each foraging decision we observed and what information was learned after making that foraging decision.

B. *Participants*

The participants were computer science students with software engineering experience. All participants had 3–7 years of programming experience (mean: 4 years). All had 1–7 years of Java programming experience (mean: 3 years), and all were familiar with Eclipse. They were 20–30 years of age; 9 were males and 2 were females.

C. *Qualitative Analysis Methods*

We used a multi-phase qualitative coding process to analyze participants' information foraging behavior, as depicted in Fig. 3. For each coding phase, after two researchers developed and refined the rules for each code set, they independently coded the same 20% of the data (at least). We then calculated inter-rater reliability using the Jaccard index. Our inter-rater reliability was 81%–92% on all code sets. Given this high level of reliability, the two researchers then split up the remaining data to code independently.

1) Code set A: Participants' foraging decisions (RQ1–4)

We first focused on participants' *foraging decisions*—the moments they explicitly chose one patch over another (Fig. 3A-i). We coded a foraging decision (1) if the participant verbalized that he/she was making a decision between visiting two or more patches (e.g., Java methods), (2) if the participant's mouse movement included hesitation over two or more hyperlinks from a list (e.g., as in search results), or (3) if the participant scrolled between two or more methods while deciding which to investigate next.

We then verified our decision codes using the retrospective interviews (Fig. 3A-ii). Specifically, after we coded the foraging decisions from the videos, we then checked what each participant had said during the interview. If he/she stated that no foraging decision had occurred in a place where we had a coded one, we removed the code. If during the interview, the participant pointed out a foraging decision that we had not coded, we added it. One example was when a participant paused to consider which method to investigate next without speaking or moving the mouse. If the interviewer did not ask a participant about an instance that we later identified as a foraging decision, we let our code stand.

2) Code set B: Participants' information goal types (RQ1)

For each foraging decision, we then coded the participant's information goal type (Fig. 3B). To identify participants' information goals as they foraged, we coded participants' information goals using 44 previously documented questions developers ask [38] (Fig. 3B-i). For the purpose of analysis, we used the four categories that Sillito et al. grouped the questions into. We chose this code set because it was a good fit for the program-debugging domain and was consistent with information goals reported in other studies [9][19][28][30]. We then mapped the goals to corresponding foraging decisions whenever participants said that a foraging decision was connected to a particular information goal (Fig. 3B-ii).

3) Code set C: Participants' cues (RQ3)

For each foraging decision, we also coded the types of cues the participant considered when making the decision (Fig. 3C). Recall that a cue acts as a signpost, providing hints as to the information at the end of a link. Since no existing cues code set was available, two researchers iteratively developed coding rules for the types of cues to which participants attended based on their verbalizations. That code set is detailed with the RQ3 results (Section IV.C) for clarity of presentation.

4) Code set D: Participants' foraging successes (RQ4)

We coded the outcome of a foraging decision as successful if the participant said that his/her information goal was

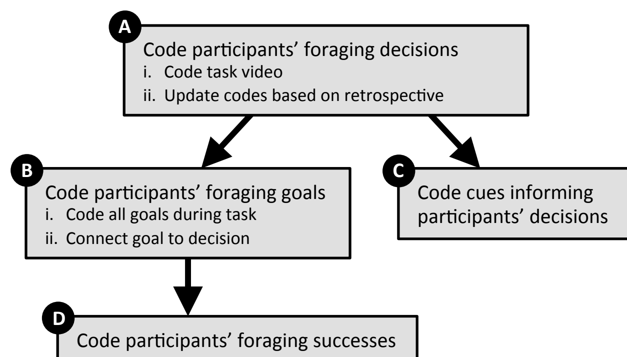

Fig. 3. Our multi-phase qualitative coding process.

fulfilled, as unsuccessful if the participant said it was not, or as unknown if the participant gave no indication (Fig. 3D).

5) Objective categorizations

In addition to the above subjective code sets, we were able to objectively derive (thus with 100% reliability) two categorizations directly from the data: patch types (RQ2) and foraging tactics (RQ4). We detail these categorizations in their associated results sections.

D. Statistical Analysis Methods

Although it is atypical to statistically analyze qualitative talk-aloud data, we were able to obtain enough data to allow quantitative analysis as well. Our statistical analysis investigated whether participants in the two groups had different information *goals* (RQ1), made decisions in different *patch* types (RQ2), relied on different *cues* (RQ3), and followed different *tactics* (RQ4).

The simplest analysis approach would have been to use chi-squared tests—for example, to construct a two-factor table of treatment group versus patch type, compute for each table cell the number of decision events from the corresponding treatment group and patch type, and then to use the chi-squared test to test the null hypothesis that treatment had no effect. Unfortunately, this simplistic approach would fail to account for the fact that, within a given participant, navigation events are not statistically independent.

Therefore, we instead relied on a three-factor table of the participant identity, the treatment group, and the IFT factor of interest (i.e., goal/patch/cue/tactic). Since the chi-squared test does not apply to three-factor designs, we used the well-established method of log-linear transformation followed by analysis of residual deviance [1], which yields a chi-squared statistic suitable for computing a valid p value. Even though we had only 11 participants, we had adequate statistical power because our unit of analysis with this technique is not participants, but rather decision points (81 total).

IV. Results

We report the results for each research question in turn. In the following, P*ng* denotes the participant with ID *n* in treatment *g* (e.g., P9F is Participant #9 in the Fix group).

A. RQ1. What Shall I Look For? Participants' Goals

For RQ1, participants' information goals, we coded their goal-related verbalizations around each decision point (code sets A and B in the previous section). Statistically, Fix and Learn participants' information goal types did not differ significantly (Analysis of deviance, $\chi^2(4)=7.53$, p=0.11). Qualitatively, the Fix participants' and Learn participants' goals also seemed similar. For example, participants in both treatments looked for the "delete lines" menu-item:

> P2F: I'm going to search for that delete lines thing in the code to see what it does.
> P10F: So I can search for that text. *Delete lines*.
> P4L: Delete lines. I'm searching for delete lines.
> P11L: Okay I should be looking for this text (*delete lines*). How do I find this text?

This lack of a statistical difference in information goals suggests two possible interpretations. If there was in fact no difference, then differences between the groups' foraging behavior reported in the upcoming sections occurred *despite* there being no difference in the participants' information goals. On the other hand, if there was a marginal difference, then it *adds* to differences reported in the upcoming sections.

B. RQ2. Turning Points: The Patches from which Participants Made Navigation Decisions

A foraging decision among multiple cues is a turning point: should I go to A to fix something, to B to learn something about the code, or to C to learn something different? To address RQ2, we analyzed the patches at which each of these turning points occured.

Toward that end, we operationalized IFT's patch construct such that each view (sub-window) in an Eclipse window and each jEdit window (i.e., the program with the bug) was a *patch type*, containing one or more patches. For example, Fig. 1 depicts an Eclipse window with three patch types, each containing at least one patch .

Table 1 shows the types of patches and characterizes the content each type offers. For example, Package Explorer patches (e.g., Fig 1a) give high-level structural overviews of the code, whereas Stack Trace patches (e.g., Fig. 4) give links to low-level code locations along with an execution path that produced an error (i.e., a thrown exception), and Search Result patches (e.g., Fig. 5) give a list of navigable search results. Table 1 only shows patches within the IDE due to a lack of foraging outside of Eclipse. (The exceptions were P2L, P5L, P9F, who briefly used a web browser.) Patches may also exist elswhere, but we did not include them in Table 1.

With these patch types, we analyzed participants' navigation decisions (Section III.C.1) among the patches. Because each navigation decision by a forager had a start and a destination, there are two patches of potential interest for each decision: the starting patch and the destination patch.

For example, a programmer might be reading in the Package Explorer view (Fig. 1a), and make a navigation decision, clicking on one of the hyperlinked items in the view. As a result, the Editor (Fig. 1b) would automatically open a file and scroll to display a particular line of code. In this example, the starting patch was in the Package Explorer and the destination patch was in the Editor.

The links provided by Eclipse views predominantly lead to Eclipse's editor view (as destination), so it is not surprising that most (76%) of navigation decisions led *to* the Editor, regardless of treatment group. So the two treatment groups did not differ in their decisions' *destination* patch types.

However, as shown in Fig. 6, Fix and Learn participants demonstrated significant differences in the *starting* patch types, i.e., those *from* which they made navigation decisions (Analysis of deviance, $\chi^2(11)=28.8$, p=0.002).

TABLE 1. TYPES OF PATCHES IN WHICH PARTICIPANTS MADE FORAGING DECISIONS. THE RIGHTMOST COLUMN IS THE NUMBER OF FORAGING DECISIONS MADE FROM THAT PATCH TYPE.

Patch Type	Information and Navigational Links	#
Editor	Provides a listing of the code in a file. Identifiers in the code are linked to associated Call Hierarchy and/or Search Results patches. Example in Fig. 1b.	37
Stack Trace	Provides a list of code locations along an execution path that produced an exception (i.e., internal error) in the running jEdit program. List items are linked to the associated lines of code (opened in an Editor patch). Example in Fig. 4.	23
Package Explorer	Provides a hierarchical list of the components (e.g., packages, classes, fields, methods) in the project. List items are linked to associated lines of code (opened in an Editor patch). Example in Fig. 1a.	18
Search Results	Provides a generated list of occurrences of user-entered text or a user-selected identifier in an Editor patch. List items are linked to the associated lines of code (opened in an Editor patch). Example in Fig. 5.	18
Call Hierarchy	Provides a hierarchical list of the invocations of a programmer-selected method (i.e., subroutine). List items are linked to the associated lines in code files (opened in an Editor patch).	3
jEdit Running Instance	Provides the interface of the running jEdit program being worked on. Such patches do not provide any direct navigational links to other types of patches.	2
Open Resource	Provides a list of all classes in the project filtered and sorted based on a programmer-entered text query. List items are linked to the associated lines of code (opened in an Editor patch).	2
Outline View	Provides a hierarchical list of the components (e.g., classes, fields, methods) of the file in the Editor patch. List items are linked to the associated lines of code (opened in the Editor). Example in Fig. 1c.	2
Variables View	Provides a list of variables and associated values at a given point in the execution of the jEdit program. Such patches do not provide any direct navigational links to other types of patches.	1

```
playManager.getNextVisibleLine(DisplayManager.java:162)
ollLineCount.reset(ScrollLineCount.java:57)
playManager.notifyScreenLineChanges(DisplayManager.java
ferHandler.doDelayedUpdate(BufferHandler.java:384)
ferHandler.transactionComplete(BufferHandler.java:336)
Buffer.fireTransactionComplete(JEditBuffer.java:2360)
Buffer.endCompoundEdit(JEditBuffer.java:2113)
```

Fig.4. Example Stack Trace patch (Fix Participant P3F).

```
jedit
  browser
    BrowserCommandsMenu.java (12 matches)
      50: files[0].getDeletePath();
      56: delete open files from the favorites. */
      57: boolean deletePathOpen = (jEdit.getBuffer(files[0].getDelete
      59: boolean delete = !deletePathOpen
```

Fig.5. Example Search Results patch.

Specifically, Learn participants tended to make navigation decisions in patches that were like the table of contents (ToC) of a book: Package Explorer and Search Results patches (e.g., Fig. 1a and Fig. 5, respectively). These patches were ToC-like because the information they contained described the hierarchical structure of the code components (i.e., the chapters and sections of the book), and the links connected to code elements at the granularity of components. Although Search Results patches might not immediately seem ToC-like, their results were presented as a structural hierarchy similar to Package Explorer patches (i.e., with hierarchical package and class file nodes that participants had to expand to reveal their contents).

In contrast to the Learn participants, Fix participants tended to make navigation decisions in patches that were like the index of a book: Stack Trace patches (e.g., Fig. 4). Like an index, Stack Trace patches contained a flat (non-hierarchical) list of elements, and their links connected to individual lines of code. Unlike Search Results and Package Explorer patches, these index-like patches provided few cues regarding how destination patches were embedded within the context of the overall code structure.

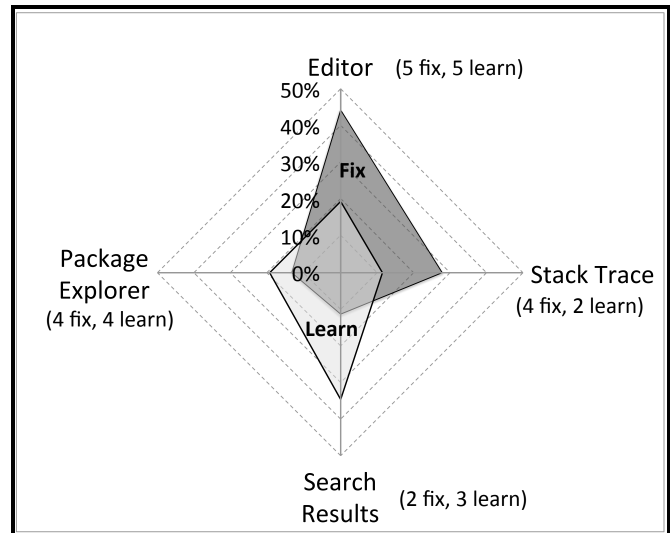

Fig.6. The proportion of patches where each treatment's participants made foraging decisions (showing the 4 most-often used patch types). Percentages indicate the number of navigation decisions in a patch type divided by the total number of navigation decisions (per-treatment). In parentheses is the number of participants in each treatment that made at least one navigation decision in that patch type.

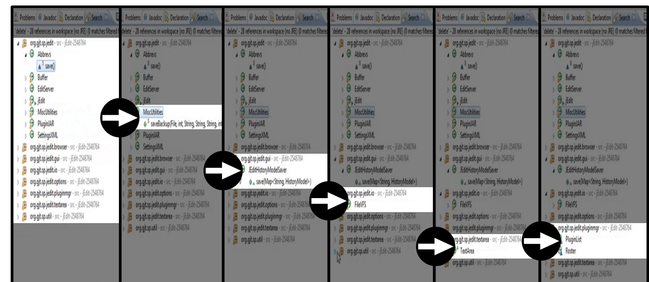

Fig.7. Learn Participant P5L engaged in considerable within-patch foraging for code-structure information using a ToC-like Search Results patch. Each box from left to right shows a snapshot of the patch. The highlighted areas show the sequence of sub-items that P5L expanded.

For example, consider two participants: Learn Participant P5L and Fix Participant P3F. P5L foraged extensively within a ToC-like Search Results patch. As Fig. 7 depicts, he stepped sequentially through the code-hierarchy tree, expanding many elements to reveal their inner structure. In contrast, P3F made considerable use of an index-like Stack Trace patch for between-patch foraging, as Fig. 8 illustrates. Whereas P5L was concerned with the code's hierarchical structure, P3F preferred to bypass the structure, linking into the middle of structural components to inspect individual lines of code.

P3F in fact commented explicitly on a desire to avoid certain types of patches…

> P3F: Since [the bug is] something that I'm trying to fix in a hurry, I would prefer to do as little of that really high level architectural stuff.

Implications for tools: When developing a new tool, there is a temptation simply to display its results as a new view within the IDE (such as a new View in Eclipse), but our results suggest that this decision should not be made lightly. If the information is hierarchical (i.e., ToC-like), such a new view could be a good fit for developers while they are in "learn" mode, On the other hand, developers in "fix" mode might be better served by the tool *enhancing* code-level patches with its new information. For example, in Eclipse, this might be achieved adding iconography within the source code editor.

Implications for theory: The Learn participants' tendency toward patches that convey high-level structure suggests that they assessed cost/benefit differently than Fix participants did. Participants assigned to the Learn treatment might have associated a higher level of benefit to these patches, versus the Fix participants who might have associated more benefit with getting to the code-level patches where they would need to make edits in order to fix the bug. Our results suggest the need for extending IFT with more detailed scent models that more precisely take into account the developer's particular situation.

C. RQ3. Turning Point "Why"s: Cue Types as Sources of Inspiration

A turning point in navigation is not likely to occur in a vacuum: something the developer has seen often inspires their ensuing decisions. Thus, for RQ3, the cue types we coded (Table 2) pertain not to the content of the cues, but rather, to the *source of inspiration* causing the participant's attention to be drawn to that cue.

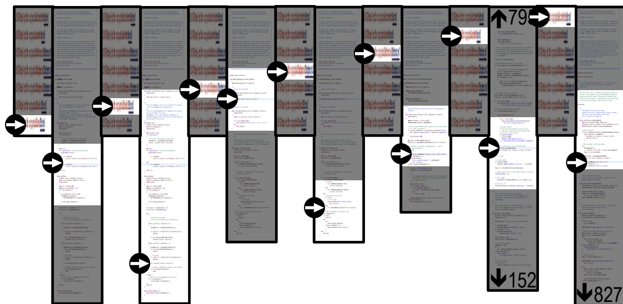

Fig.8. Fix Participant P3F used the Stack Trace patch from Fig. 4 as an index to forage for particular lines of code in a variety of Editor patches. Each box from left to right shows a snapshot of a patch, alternating between the Stack Trace patch and Editor patches. The arrows show the links from the Stack Trace Patch and the destination lines of code within the Editor patches. In the Editor patches, the shaded areas were never visible to P3F (i.e., off screen).

For example, as Fig. 9 shows, P5L made a navigation decision in an Editor patch (c), and as he did so, indicated that his attention was on cues that were "something like a click on a menu button" (d). The source for this inspiration came from a preceding visit to a jEdit Output patch (a) in which he used the jEdit menu item to trigger a failure caused by the bug (b). Thus, the cue in this case was of type Output-inspired.

Applying these cue types, our results revealed marked differences in the cues to which Fix and Learn participants attended. A log-linear analysis of the cue-type frequencies showed a significant difference (Analysis of deviance, $\chi^2(19)=33.3$, $p=0.02$). Fig. 10 highlights these differences for the most-attended cue types.

One difference apparent in Fig. 10 is that the Learn participants particularly attended to Output-Inspired cues, citing Output-Inspired cues in about 45% of their decisions, compared to only 28% for Fix participants. One possible reason for this tendency was that many Learn participants followed a bug-reproduction-driven approach in which they followed up on the application's expected or observed output from the very outset of their session. For example, Learn participants P6L, P7L, and P11L all began by replaying the error in jEdit, and then attending to cues inspired by the

TABLE 2. THE CUE TYPES TO WHICH PARTICIPANTS ATTENDED WHEN MAKING FORAGING DECISIONS. THE RIGHTMOST COLUMN IS THE NUMBER OF FORAGING DECISIONS IN WHICH PARTICIPANTS ATTENDED THAT CUE TYPE.

Cue Type	Definition: Participant utterances about…	#
Output-Inspired	… cues related to jEdit output they had seen, such as thrown exceptions (errors) or GUI widget labels	44
Domain Text	… cues related to text they had seen specific to jEdit's domain, such as "folding text"	35
Level of Abstraction	… cues related to the level of abstraction of the a code location they had seen; e.g., "This method is too specific"	21
Source-Code Content Inspired	… cues related to source code they had seen, such as relating to a particular variable or parameter, or reminiscent of a code comment	19
Position	… cues related to the position of non-code elements they had seen on screen, such as the top item in a list of search results	8
Familiarity	… cues that seemed familiar to the participant; e.g., "I've seen this before" or "This looks familiar".	6
File Type	… cues related to the type of a file they had seen, such as Java vs. XML	5
Documentation-Inspired	… cues related to external documentation they had seen, such as the bug report	2
Source-Code Appearance Inspired	… cues related to how source code they had seen appears visually, such "large" methods or "nearby" methods	2
Contrasts	… cues related to a contrast among items they had seen, such a method being from a different package than the others in a list	2

relevant GUI elements (e.g., menu items). Although Fix participants also sometimes attempted to reproduce bugs, they did so much less than Learn participants.

The sources of output that inspired Fix and Learn participants also differed. Whereas Learn participants primarily found inspiration in the program's visual output (i.e., from running jEdit), Fix participants predominantly attended to Output-Inspired cues that originated in a Stack Trace patch or Variables patch, which were more closely related to low-level code details. In fact, only one Fix participant, P2F, had Output-Inspired cues that were inspired by running jEdit.

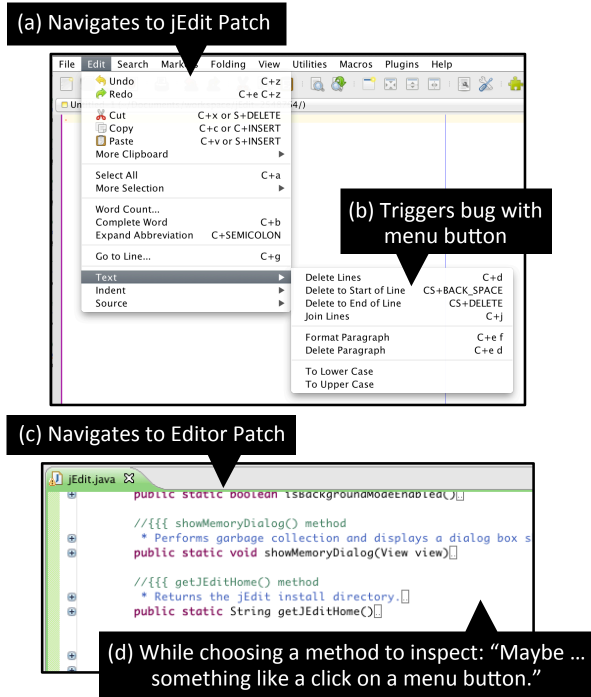

Fig.9. Episode in which P5L attended to an Output-Inspired cue. The source of the inspiration came from jEdit output (a and b). P5L attended to the Output-Inspired Cue ("Maybe … something like click on a menu button") while choosing code to inspect in an Editor patch (c and d).

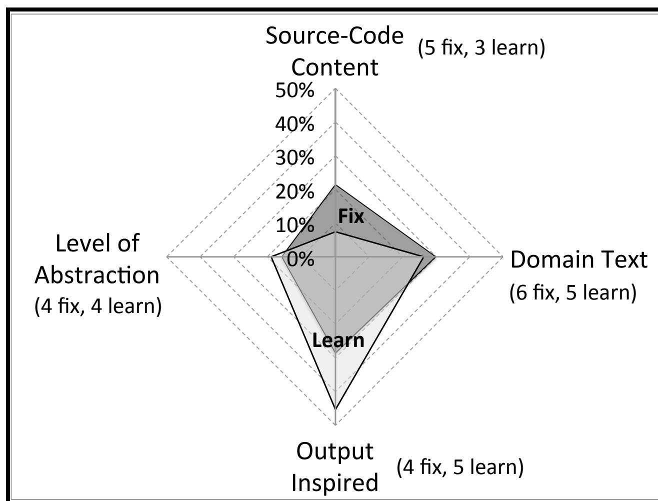

Fig. 10. The proportion of cue types each treatment's participants attended to at each navigation decision, showing the four most-often attended cue types. Percentages indicate the number of cue types attended divided by the total number of cue types attended to (per treatment). In parentheses is the number of participants in each treatment that attended to that cue type at least once. Fix and Learn participants attended to different cue types for foraging decisions. Shown are the percentages of each group's total number of cue types talked about when they decided among competing cues for the most mentioned cue types.

Fix participants' foraging stayed "closer to the code" than that of Learn participants in two additional ways, as well. First, as shown in Fig. 10, Fix participants attended to Source Code-Content Inspired cues more than Learn participants (approximately 21% for Fix participants versus 8% for Learn participants). Second, all but two of the instances when Fix participants attended to this cue type occurred in an Editor patch. In contrast, Learn participants attended to Source Code-Content Inspired cues over a wider spread of patch types, including high-level patch types such as Package Explorer and Outline View, as well as low-level patch types like Editor and Stack Trace. Thus, although both treatments sometimes attended to cues inspired by source code they had seen, Fix participants attended to those cues mainly while the source code, but Learn participants tended to those cues more broadly, even in patches not directly related to the details found in the source code.

Implications for tools: These results highlight the fact that developers might benefit from different cue-enhancement tools depending on whether they are trying to fix or learn about bugs. For example, the behavior of developers in the Learn treatment suggests that they might especially benefit from tools oriented toward augmenting Output-Inspired cues. An example of such a tool is the Whyline [13], which offers annotated, navigable links directly from program output to the code that generated that output. On the other hand, the behavior of developers in the Fix treatment suggests that they might particularly value tools that augment Source Code-Content Inspired cues. For example, a tool for Eclipse might automatically annotate the code with annotations or links related to associated bug reports (e.g., as proposed in [26], figure 3).

For both Fix and Learn developers, Level of Abstraction and Domain Text cues played an important role. Most IDEs and languages already provide support for viewing the level of abstraction associated with a particular piece of code (method, class, package, etc.) and for navigating among different levels of abstraction. Research has also investigated how to map from concerns (i.e., domain requirements such as jEdit's text-folding functionality) to specific locations in the code [24], as well as how to analyze code and automatically generate natural language text describing the corresponding concerns [33]. Our results reiterate the potential value of such tools and emphasize the need for getting them into everyday practice by developers.

Implications for theory: Our results suggest a possible new direction for IFT. Prior IFT research has often operationalized cue content through the use word-similarity metrics like TF-IDF (e.g., [18][28]). Our findings are consistent with this approach: several commonly attended types of cues were text based (e.g., Source Code Content Inspired and Domain Text). However, our cue types also revealed differences between the treatments *without* considering cue content. Thus, our inspiration-based cue types were able reveal effects on foraging that cue content alone might not have. Future IFT-based models of cues should take into account not only the

content of cues but also the type of these cues when generating predictions about developers' foraging behavior.

D. RQ4. How Should I Go About Foraging? Participants' Foraging Tactics

For RQ4 (Fix versus Learn participants' foraging tactics), participants used two distinct foraging tactics in deciding which cues to attend to: a *Favorites* tactic and a *Switching* tactic. Using the Favorites tactic, a participant would, from one foraging decision to the next, attend to cues sharing at least some of the same cue types. In contrast, a participant using the Switching tactic would, from one foraging decision to the next, attend to cues of entirely different types.

We counted the Favorites and Switching tactics using discrete, objective patterns in the sequence of cue types to which each participant attended. In particular, if the set of cue types attended for one foraging decision was disjoint from the set attended for the next decision, we counted it as an instance of the Switching tactic. If the sets intersected, we counted it as an instance of the Favorites tactic.

To illustrate the difference in these tactics, consider Fix participants P10F and P9F. P10F used the Favorites tactic heavily, attending to cues of Domain Text type for every foraging decision, continually asking throughout the session where "delete line" was located. In contrast, Fix Participant P9F attended to Familiarity type cues in one successful foraging decision, but then switched to the Source Code-Content Inspired and Domain Text cue types in his next foraging decision only one and a half minutes later.

At first glance, little difference was apparent between the tactic participants used (Table 3, "Most-Used" column). Most participants, regardless of treatment, tended toward the Favorites tactic. Analysis of their activity showed a suggestive, but non-significant difference between treatments (Analysis of deviance, $\chi^2(3)=5.93$, $p=0.12$).

However, a difference between treatments becomes apparent with situations where the outcome of a foraging action was *successful* (code set D in Section III.C). In these situations, Fix participants were equally likely to be successful regardless of whether they used the Favorites tactic or the Switching tactic (Analysis of deviance, $\chi^2(1)=1.10$, $p=0.29$). In contrast, Learn participants were significantly more likely to be successful when they used the Switching tactic (Analysis of deviance, $\chi^2(1)=5.49$, $p=0.019$).

TABLE 3. PARTICIPANT USAGE AND SUCCESS RATES FOR THE FAVORITES AND SWITCHING TACTICS.

Treatment	Participant	Most-Used Tactic	Success Rate	
			Favorites	Switching
Fix	P2F	Favorites	1/7	0/2
	P3F	Favorites	1/6	0/2
	P8F	Favorites	2/5	0/2
	P9F	Switching	1/1	1/4
	P10F	Favorites	0/3	0/1
	P12F	Both Equally	0/1	0/1
		Total instances of success:	5/23 (22%)	1/12 (8%)
		Total participants who had success:	4/6 (67%)	1/6 (20%)
Learn	P4L	Switching	0/2	2/4
	P5L	Favorites	1/5	0/0
	P6L	Favorites	0/4	0/2
	P7L	Both Equally	0/2	2/2
	P11L	Favorites	0/4	0/1
		Total instances of success:	1/17 (6%)	4/9 (44%)
		Total participants who had success:	1/5 (20%)	2/5 (40%)

Thus, not only did the Fix and Learn participants make foraging decisions in different types of patches (RQ2) and while attending to different types of cues (RQ3), from the RQ4 analysis in this section, we know they also differed in which foraging tactics most strongly associated with whether or not their decisions yielded success.

Implications for tools: Many studies (Section V) have highlighted the gradual but non-monotonic transition during a maintenance task from exploring code to modifying code, potentially implying a corresponding shift from behaving as if in our Learn treatment to behaving as if in our Fix treatment. It is not known whether such transitions affect the level of production bias developers feel (or, conversely, whether developers as individuals have a more permanent tendency toward learning or fixing). But, to the extent that developers do transition from one mode to the other during a maintenance task, then they may benefit from a corresponding transition in tactics.

Given our result that Switching among cue types was more strongly associated with success in the Learn treatment than in the Fix treatment, it is possible that it would be beneficial for tools likewise to emphasize a diversity of cue types early in a development task. For example, when new prototype tools generate search results or navigation recommendations such as those described in [12] and [22], it might be beneficial for them internally to rely more heavily on a diversity of cue types early in a maintenance task, then gradually transition toward the code-level cues that developers preferred in our Fix treatment.

Providing an "intelligent" tool that attunes itself to the stage of the maintenance task would require detecting a developer's potentially gradual transition from learning to fixing. For example, could an IDE detect the transition based on the developer's initiation of non-trivial code edits? Having detected this transition, a tool might subtly and intelligently transition itself from learning-support mode to fixing-support mode. How a tool could effectively make such adaptations, yet also allow developers to access things the tool had not foreseen, is an interesting open question.

Implications for theory: IFT's central proposition is framed in terms of the singular decision that a developer faces at a particular moment (Section II): should the developer stay in the current patch, navigate to another patch, or enrich the environment? Our results suggest the need for modeling of foraging at a higher level that spans *sequences* of foraging decisions, since we found that Learn participants were more likely to succeed by switching among different cue types at each decision point. Such research could investigate not only what tactics are employed relative to different cue types but also "meta-tactics" aimed at selecting an optimal tactic for a given situation in time (e.g., depending on whether the forager is currently trying to learn about or trying to fix a bug).

V. RELATED WORK

Studies on how developers forage for code have typically focused on the kinds of questions that they ask, or the kinds of information that they try to obtain (e.g., [16][38][41]). A recurring theme from these studies is that of progression, with developers first trying to find a starting point in the code related to a bug or feature request, then expanding their search outward as they generate and test hypotheses about how the code works and how to go about fixing it. The segments of code that trigger hypotheses and subsequent searches for further information are sometimes called "beacons" [4], and the succession of searches can lead to a bottom-up or top-down comprehension of the code, depending on the programmer [40]. Once convinced that they have a viable approach for fixing the code, they make and validate the code changes. Thus, these prior studies have largely focused on information goals and changes in goals during the course of a maintenance task. In contrast, our study focused on how production bias affects developers' foraging at turning points among competing paths.

Other studies have examined the strengths and weaknesses of existing IDE tools or have evaluated new tools that support information foraging. Many studies noted the importance of search tools and call graphs (e.g., [14][17][34][39][45]) as a means of finding information in code. Robillard et al. particularly highlighted the value of tools that aid in methodically exploring code, using structure-guided search, and planning changes prior to editing code [34]. Many recent tools and supporting algorithms have been provided specifically for the problem of locating where in code a given piece of buggy functionality is implemented. The latest approaches include integration of text and stack trace analysis [25][43], integration of search with navigation [12], and genetic algorithms [42] as well as machine learning algorithms [2][44] for dynamically refining and combining localization models. Our study's results reiterate the need for diverse new tools to aid developers, and they provide insight into the situations in which different developers might benefit the most from different tools.

Finally, several studies have investigated how people edit code once they have found the needed information. For example, Ying and Robillard examined whether developers make edits differently depending on whether they are fixing bugs or adding enhancements [46]. 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 [32]. In our study, although participants were editing at times, our research questions centered on foraging rather than on editing per se.

VI. THREATS TO VALIDITY

Every study has threats to validity, but we guarded against threats to internal validity in several ways. To help assure *content validity* (the extent to which all aspects of a theoretical concept are measured), we examined the effects of treatment group with respect to a broad range of IFT constructs (goal/patch/cue/tactic). Our inter-rater reliability was 81%–92% on all code sets, helping to assure construct validity. To help assure *test validity* (the extent to which a measure actually captures what it intends to measure), we supplemented qualitative analysis with quantitative analysis and carefully controlled for any confounding per-participant effects.

The primary threat to external validity is that our study participants were undergraduate computer science majors, although all had at least 3 years of programming experience. We can consider them a reasonable proxy for developers with fairly low experience (including interns and new hires straight out of college), but our findings might not generalize to developers with far more experience. Similarly, there is a question as to the generalizability of the tasks; however, fixing tasks are common during corrective maintenance, and software immigrants may be tasked with just learning about code [36]. Additionally, there is a question of generalizability to other programming languages and IDEs, which we defer to future work.

VII. CONCLUSION

The results of our empirical study show, for the first time, how Minimalist Learning Theory's concept of production bias can influence developers' foraging for information. Developers engaged with fixing the bug (and doing whatever learning they needed to along the way) vs. those engaged with learning "enough" about the bug to help someone else, differed considerably in their foraging:

- *Patch Types*: Learn vs. Fix participants' turning points—should I go here or should I go there—occurred in significantly different types of patches. Learn participants tended to work their way through *hierarchical, table-of-contents-like patches* that made explicit information structure, whereas Fix participants tended to favor *low-level index-like patches* that took them directly to a line of code.
- *Cue Types*: Fix and Learn participants also differed significantly in the types of cues that drew their interest during these foraging decisions. Learn participants followed cues they had seen in program output in nearly half of their navigation decisions (45%, almost twice as often as Fix participants). In contrast, Fix participants favored following cues inspired by source code content.
- *Foraging Tactics*: Learn participants' successful tactics were different from those of Fix participants. Learn participants were more successful when switching among cue types in sequential decisions, whereas Fix participants were more successful foragers when they used the same cue types repeatedly over several decisions.

As noted in the "Implications for Tools" subsections of Section IV, these theory-based results reveal new opportunities for ways tools can better enable developers to find information during everyday software maintenance.

ACKNOWLEDGMENT

This material was supported in part by the National Science Foundation 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 sponsor.

References

[1] Agresti, A. (2012) *Categorical Data Analysis*, Wiley.

[2] Binkley, D., and Lawrie, D. (2014) Learning to rank improves IR in SE. *IEEE Intl. Conf. Soft. Maint. and Evolution*, 441-445.

[3] Brandt, J., Guo, P., Lewenstein, J., Dontcheva, M., and Klemmer, S. (2009) Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. *ACM Conf. Human Factors in Comp. Sys.*, 1589-1598.

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

[5] Carroll, J. (1998) Minimalism Beyond the Nurnberg Funnel, MIT Press.

[6] Carroll, J., and Rosson, M. (1987) Paradox of the active user. *Interfacing Thought: Cognitive Aspects of Human-Comp. Interaction*, 80-111.

[7] Chi, E., Pirolli, P., Chen, K., and Pitkow, J. (2001) Using information scent to model user information needs and actions and the web. *ACM Conf. Human Factors in Comp. Sys.*, 490-497.

[8] 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.*, 22(2), 14:1–14:41.

[9] 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*, 19(1), 5:1–5:28.

[10] 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.*, 2511-2520.

[11] 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*, 217-224.

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

[13] Ko, A., and Myers, B. (2008) Debugging reinvented. *ACM/IEEE Intl. Conf. Soft. Eng,* 301-310.

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

[15] 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.*, 3073-3082.

[16] LaToza, T., and Myers, B. (2010) Developers ask reachability questions. *ACM/IEEE Intl. Conf. Soft. Engr.*, 185-194.

[17] LaToza, T., Venolia, G., and DeLine, R. (2006) Maintaining mental models: A study of developer work habits. *ACM/IEEE Intl. Conf. Soft. Engr.*, 492-501.

[18] 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.*, 1323-1332.

[19] 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,* 39(2), 197-215.

[20] 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.*, 25-34.

[21] Lieber, T., Brandt, J., and Miller, R. (2014) Addressing misconceptions about code with always-on programming visualizations. *ACM Conf. Human Factors in Comp. Sys.*, 2481-2490.

[22] Maalej, W., Fritz, T., and Robbes, R. (2014) Collecting and processing interaction data for recommendation systems. *Recommendation Sys. in Soft. Engr.*, 173-197.

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

[24] Majid, I., and Robillard, M. (2005) NaCIN: An eclipse plug-in for program navigation-based concern inference. *OOPSLA Workshop on Eclipse Technology Exchange*, 70-74.

[25] Moreno, L., Treadway, J., Marcus, A., and Shen, W. (2014) On the use of stack traces to improve text retrieval-based bug localization. *IEEE Intl. Conf. Soft. Maint. and Evolution*, 151-160.

[26] Murphy, G., Kersten, M., Robillard, M., and Cubranic, D. (2005) The emergent structure of development tasks. *ECOOP Conf. Object-Oriented Prog.*, 33-48.

[27] Olston, C., and Chi, E. (2003) ScentTrails: Integrating browsing and searching on the web. *ACM Trans. Comp.-Human Interaction*, 10(3), 177-197.

[28] 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.*, 1471-1480.

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

[30] Pirolli, P., and Card, S. (2005) The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. *Intl. Conf. Intelligence Analysis*, 2-4.

[31] Pirolli, P., Schank, P., Hearst, M., and Diehl, C. (1996) Scatter/gather browsing communicates the topic structure of a very large text collection. *ACM Conf. Human Factors in Comp. Sys.*, 213-220.

[32] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. (2013) Dual ecological measures of focus in software development. *ACM/IEEE Intl. Conf. Soft. Engr.*, 452-461.

[33] Rastkar, S., Murphy, G., and Bradley, A. (2011) Generating natural language summaries for crosscutting source code concerns. *IEEE Intl. Conf. Soft. Maint.*, 103-112.

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

[35] Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012) How do professional developers comprehend software? *ACM/IEEE Intl. Conf. Soft. Engr.*, 255-265.

[36] Sim, S. and Holt, R. (1998) The ramp-up problem in software projects: A case study of how software immigrants naturalize., *ACM/IEEE Intl. Conf. on Soft. Engr.*, 361-370.

[37] Simon, H. (1972) Theories of bounded rationality. Decision and Organization. 1(1), 161-176.

[38] Sillito, J., Murphy, G., and De Volder, K. (2006) Questions programmers ask during software evolution tasks. *ACM Intl. Symp. Found. of Soft. Engr.*, 23-34.

[39] 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.

[40] von Mayrhauser, A., and Vans, A. (1996) Identification of dynamic comprehension processes during large scale maintenance. *IEEE Trans. on Soft. Engr.* 22(6), 424-437.

[41] Wang, J., Peng, X., Xing, Z., and Zhao, W. (2011) An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. *IEEE Intl. Conf. Soft. Maint.*, 213-222.

[42] Wang, S., Lo, D., and Lawall, J. (2014) Compositional vector space models for improved bug localization. *IEEE Intl. Conf. Soft. Maint. and Evolution*, 171-180.

[43] Wong, C., Xiong, Y., Zhang, H., Hao, D., Zhang, L., and Mei, H. (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. *IEEE Intl. Conf. Soft. Maint. and Evolution*, 181-190.

[44] Xuan, J., and Monperrus, M. (2014) Learning to combine multiple ranking metrics for fault localization. *IEEE Intl. Conf. Soft. Maint. and Evolution*, 191-200.

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

[46] Ying, A., and Robillard, M. (2011) The influence of the task on programmer behaviour. *IEEE Intl. Conf. Prog. Comprehension*, 31-40.