Foraging Goes Mobile:

Foraging While Debugging on Mobile Devices

David Piorkowski[1], Sean Penney[2], Austin Z. Henley[3], Marco Pistoia[1], Margaret Burnett[2], Omer Tripp[4*], Pietro Ferrara[5*]

[1]IBM Research
Yorktown Heights, NY, USA
david.piorkowski@ibm.com
pistoia@us.ibm.com

[2]Oregon State University
Corvallis, OR, USA
{penneys, burnett}
@eecs.oregonstate.edu

[3]University of Memphis
Memphis, TN, USA
azhenley@memphis.edu

[4]Google
New York, NY, USA
trippo@google.com

[5]Julia SRL
Verona, Italy
pietro.ferrara@juliasoft.com

***Abstract*—Although Information Foraging Theory (IFT) research for desktop environments has provided important insights into numerous information foraging tasks, we have been unable to locate IFT research for mobile environments. Despite the limits of mobile platforms, mobile apps are increasingly serving functions that were once exclusively the territory of desktops—and as the complexity of mobile apps increases, so does the need for foraging. In this paper we investigate, through a theory-based, dual replication study, whether and how foraging results from a desktop IDE generalize to a functionally similar mobile IDE. Our results show ways prior foraging research results from desktop IDEs generalize to mobile IDEs and ways they do not, and point to challenging open research questions for foraging on mobile environments.**

Keywords—information foraging theory; mobile software development

I. INTRODUCTION

Mobile devices are slowly taking over functionalities that once were possible only on desktop/laptop computers. Besides traditional mobile apps such as text messaging, accessing email, and navigation, mobile devices now allow complex applications such as document readers, text editors, spreadsheets, and even integrated development environments (IDEs). Where there is a complex information space (such as with programming), there is likely to be foraging, that is, information seeking behavior. However, we have been unable to locate research into how people forage on mobile devices, especially in the context of a complex information space like using a mobile IDE to debug a mobile app.

The very idea of using an IDE on a mobile device, such as a smart phone or tablet computer, may seem strange at first, because the amount of information a user must deal with may seem too complex to be displayed properly on the screen of a mobile device. However, in recent years, new IDE tools have become available that allow developers to write and debug mobile apps directly on mobile devices (e.g., [46]). This allows a tight coupling between editing and running, which affords liveness in mobile programming [4, 5, 26, 45, 48]. Liveness in turn supports rapid development, testing, debugging, configuration and deployment of mobile apps "on the go" without the need for emulators. According to one study, programmer productivity is significantly enhanced when developers are offered the opportunity to write mobile applications directly on mobile devices—particularly when writing *small* applications [29].

Perhaps programmers of larger mobile applications could also reap such benefits, if their foraging was supported on mobile devices. Thus, in order to investigate where foraging a complex information space on a mobile platform is the same and where it differs from desktop foraging, we conducted a dual replication study. Mobile and desktop platforms are different, so our study is the type known as a conceptual replication, which "test<s> the *same fundamental idea* or hypothesis behind the original study, but the operationalizations of the phenomenon, … may all differ" [9] (italics added). For our case, the fundamental idea was information foraging in IDEs. As we shall see, IFT's theoretical constructs were critical to our ability to replicate and compare findings across different platforms.

The first study we replicated investigated the interplay between learning and "doing" in desktop programmers' foraging [35]. When foraging through code, programmers sometimes need to learn about code they have stumbled upon, and sometimes they need to change the code they are in. The two goals are intertwined, but Minimalist Learning Theory (MLT) [11] predicts that the learning part of such tasks is sometimes given less attention than the doing (changing) part. Our replication study investigates to what extent the findings of learning vs. doing in a desktop IDE [35] generalize to mobile IDEs:

RQ1: Do desktop-based findings about production bias in information foraging generalize to a mobile environment?

RQ2: If so, in what ways do these findings generalize? When they do not generalize, why not?

To delve more deeply into the results of RQ1 and RQ2, we also replicated another desktop study [36] that had itself been a partial replication of the main study in that it used the same platform and study type. That study had derived a set of research challenges for supporting desktop programmers' foraging efforts that might help to explain our RQ1 and RQ2 results. Therefore, we also investigated whether those research challenges were the same for a mobile environment:

RQ3: In what ways do research challenges for supporting desktop foraging generalize to a mobile environment?

II. BACKGROUND AND RELATED WORK

Our investigation is framed by Information Foraging Theory (IFT). IFT provides a conceptual framework for explaining how people in an environment seek information.

This work was supported in part by NSF under Grants 1302113, 1302117, and 1314384 and by DARPA under Contract N66001-17-2-4030. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, the Army Research Office, or the US government.

*All of this author's contribution to this work took place while the author was employed by IBM Corporation at IBM's TJ Watson Research Center.

978-1-5386-0443-4/17/$31.00 ©2017 IEEE

The main constructs of IFT are a *predator* (a human seeking information) and *prey* (the information the predator is looking for) [37]. The predator forages through an information environment consisting of *patches* (areas containing information) and navigates between patches via *links* (connections between the patches). Each patch contains *information features*, which may contain the prey the predator is looking for. Some information features are *cues*: information features that have a link. Cues inform the predator of what information lies in the patch at the other end of the link. Fig. 1 illustrates these concepts, and Table 1 summarizes them.

Predators build *scent* by assessing cues in the environment. This assessment of cues is central to IFT's main proposition, which says that the predator treats foraging as an optimization problem. More specifically, according to IFT, the predator's foraging actions try to maximize the value V of information they will gain from a patch per their cost C of getting to and interacting with that patch, i.e., $\max(V/C)$.

However, since predators do not have perfect information about a patch's value and cost, they make their choices based on their *expectations* E of value V and of cost C, i.e., $\max(E(V)/E(C))$. The predator bases such expectations on whatever information they have gathered so far, such as by inferring them from available cues.

However, information seeking is often not the main task, but rather a necessity to complete another task. These situations are well described by Minimalist Learning Theory (MLT) [11]. MLT explains motivations and behavior of "active users"— people in situations where learning is motivated solely by another task. According to MLT, active users' focus on throughput (finishing their task as quickly as possible) leads to a motivational paradox named *production bias*. This paradox describes how active users' focus on throughput reduces their motivation to spend time learning about the task, even if doing so might help them complete the task. Several studies have reported programmers behaving in this "active user" fashion, in which they favor "doing" over learning [3, 18, 23, 40]. In this paper, we investigate how IFT and MLT come together for programmers in mobile vs. desktop IDEs.

IFT alone (without MLT) has been studied in a variety of desktop computing settings, most commonly Web foraging [6, 7, 12, 8, 14, 38, 42] and foraging in desktop programming environments [3, 16, 17, 19, 20, 21, 44]. The latter group is the most relevant to our current study.

In desktop programming, IFT has been able to model programmers' foraging to predict programmer navigation [19, 21, 32, 41], to understand developers' goals and strategies [33], to explain how tools affect programmers' navigation behaviors [16], to identify cues in Web mashups [17], to improve requirements-tool tracing [31], and to explain programmers' foraging behavior among code with multiple variants [44]. These and similar studies have led to IFT-informed approaches to support programmers' foraging [13, 27, 30, 34].

On the other hand, for mobile programming, research on *building* mobile IDEs is still emerging. Among the IDEs for a mobile device is TouchDevelop [46, 47]. TouchDevelop has been used to build a variety of applications, particularly games for smartphones and tablets [2]. There is also emerging work on providing higher-level "programming" environments that allow stitching Web services together [10] and on providing scaffolding for new Java programs [25]. Developers have also built programming tools for mobile devices, and a number of mobile IDEs are now available on the Google Play Store (https://play.google.com) (e.g., AIDE, Terminal IDE, CppDroid, AWD, DroidScript, Anacode IDE, Deuter-IDE, AIDE Web, APDE, and C++ Compiler IDE).

So far there is only a little empirical work on how developers use mobile tools like this, but there is some formative empirical work. For example, Li et al. observed that developing code directly on mobile devices has changed the way code is written [22]. Their study points out new code-development evolution patterns, such as dense external method calls and a high code-reuse ratio. The study also showed the need for a version control and code-search support for mobile IDEs to better manage existing codebases written by end users [22]. Nguyen et al. observed that debugging mobile applications is a hard problem, as current debugging techniques either require multiple compute devices or do not support graphical debugging. To address this problem, they developed Graphical On-Phone Debugger (GROPG) and showed that GROPG lowered overall debugging time while improving user experience [28]. However, none of the studies we have been able to locate have investigated how developers forage on mobile IDEs.

III. METHODOLOGY

To compare what kinds of foraging behaviors would seem to fundamentally differ between desktop/laptop and mobile environments, we replicated two previous desktop foraging talk-aloud studies [35, 36], in which participants worked on a text editor(JEdit) using the Eclipse IDE on desktops.

The first of these (the "original study") [35], was our primary source of replication. This study enabled us to investigate RQ1 and RQ2—whether and how previous findings on the interaction of production bias with foraging might generalize to, or change in, mobile environments. Its participants

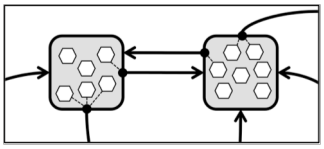

Fig. 1: A portion of an information environment containing two patches (rounded squares), connected via links (arrows). Patches contain information features (hexagons), some of which are cues (hexagons connected to links).

TABLE 1: IFT'S CONSTRUCTS AS OPERATIONALIZED IN THIS STUDY

Construct	Operationalization
Predator	The programmer seeking information
Prey	The information the programmer is seeking
Patch	Contents of view in the IDE, such as the Editor or File Explorer, but also a method or class
Link	A way to navigate to another patch, such as a clickable link in search results or scrollbar
Info. Feature	A patch's contents, such as comments, code or files
Cue	Text or other decorative elements related to a link, such as the name of a method

were 11 Computer Science students with 3–10 years of professional experience. The second study (the "Desktop value/cost study") we replicated [36] enabled investigation of RQ3. That study identified value and cost-related foraging challenges faced by programmers debugging. The participants were 10 professional developers at Oracle, with 2–19 years of professional experience. Replicating this study enabled us to address RQ3.

We use the following shorthand to differentiate participants: P*nt-E* denotes participant with ID *n*, in treatment *t*, in environment *E*. For example, P2F-M is Participant #2, in the Fix treatment, in the Mobile environment. (Mobile refers to this paper's study and Desktop refers to either of the two previous studies.)

A. Participants and Procedures

Our Mobile participants were 8 professional developers, ages 20–55, at IBM (7 males and 1 female). None of them were familiar with the study's IDE or application, but they were all experienced Android developers, and were given a tutorial of AIDE prior to the task. They had 0.5–5 years (mean 2 years) of experience with Android, 2–20 years (mean 12 years) with Java, and 1–38 years (mean 19 years) with software development.

Like the original study [35], we randomly divided the 8 participants into two treatments: Fix and Learn. Both treatments worked on the same bug. We told Fix participants to fix the bug, whereas we told Learn participants to "learn enough about the defect such that they could onboard a new programmer to fix it." Both treatments could proceed however they desired, e.g., access the Web or use other tools.

We began each session with a brief introduction explaining the study, followed by a background questionnaire. Participants had 30 minutes to complete their assigned treatment's task (the "working session"), talking aloud as they worked. We recorded the tablet's screen, as in the original study, and audio of their utterances while they worked.

Participants used a high-end tablet, the Samsung Galaxy Tab S. The programming environment was AIDE, a full-featured IDE for Android with over 2 million downloads. Fig. 2 shows a screenshot of AIDE in action. AIDE explicitly supports external keyboards, so participants used a keyboard for typing code and searching, but used the touchscreen for actions such as scrolling and selecting.

Participants from both treatments were asked to talk aloud as they worked on Vanilla Music issue #148 (https://github.com/vanilla-music/vanilla/issues/148). Vanilla Music is a mature open-source music player for Android, with between 500,000 and 1,000,000 downloads. It is written in Java with 67 classes and 13,369 non-comment lines of code. Issue #148 described a problem with playback.

After their working session, we conducted a semi-structured interview (the "retrospective session"). A researcher played back the recording of the participant's working session, and asked 6 questions about each navigation (any transition to a file or method). For the first three, we stopped the video just prior to the navigation and asked: (1) why the participant chose to navigate there, (2) what expectations they had of what they would find and (3) whether they had considered alternative navigations. We then resumed the video and, just prior to the next navigation (i.e., navigating away), we asked: (4) whether their expectations had been met, (5) what they had learned at that location, and (6) whether what they learned caused them to change their course. Like the working session, the retrospective was video-recorded.

B. Analysis Methods

We performed three types of analyses: algorithmic counting when data categorizations that did not require human judgment (e.g., *navigations*). We counted Desktop navigations whenever a participant performed an action that moved the cursor to a new class or method, and Mobile navigations when two conditions were met: (1) a new method or file was visible in AIDE's editor, and (2) the participant indicated interest in the method, either by reading it aloud or hesitating in the method. The total number of navigations was 501 across 11 participants for Desktop, and 217 across 8 participants for Mobile. We also counted types of *patches* among which participants navigated. A complete list of the patch types is given in Table 2 in Section IV.

Secondly, we used qualitative coding [24] to categorize the cue types participants talked about as well as their verbalized expectations of a navigation. For cue types, we achieved exactly the same inter-rater reliability of 83% over 20% of the data for both environments as measured by the Jaccard index. For expectations, we coded responses as either *met expectations*, *did not meet expectations*, or *unclear*. Using the Jaccard index, we achieved an inter-rater reliability of 100% on 20% of the data for this coding. Given this high level of reliability, the two researchers then split up the remaining data to code independently.

Finally, we used inferential statistics where we had enough data to statistically analyze our question of replicating the significant difference found in the original study between the Fix vs. Learn conditions. The standard chi-squared test would assume that our navigation events are statistically independent, so instead we used the well-established method of log-linear transformation followed by analysis of residual deviance [1]. Even though we had only 8 participants, we had adequate statistical power for this because our unit of analysis with this technique is not participants but rather cue and patch data points (155 and 217, respectively). One Desktop partic-

Fig. 2: AIDE in its debugger mode. Left: the currently paused stack trace. Right: the currently executing line of code is highlighted.

ipant was removed as an outlier due to 112 out of 119 navigations in the Debug patch coming from that participant, and this participant was confused about Debug functionality.

IV. Results

A. RQ1 Results: Production Bias Goes Mobile

The original study found significant differences between Fixers and Learners so we begin by analyzing such differences in the Mobile world.

1) Production Bias: Present on Mobile?

According to IFT, foragers remain in the same patch until the value/cost balance has tipped—i.e., they decide that they will obtain more value per effort by going to another patch.

In IDEs, the locations of such tipping points (patches) can be grouped into *patch types*. On the original Desktop study, 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. Following this approach, Fig. 2 depicts an AIDE window with two patch types. The Stack Trace patch type contains one patch (the active stack trace), and the Editor patch type is likely to contain multiple patches (multiple methods or classes). Table 2's left columns list the patch types in the Mobile environment, with their usage in the right column.

We can similarly group the cues influencing these navigations into cue types. This grouping abstracts individual cue contents (words) to the *source of inspiration* that drew the participant's attention to that cue. Table 3's left columns list the cue types that participants attended to, with their overall popularity in the right column.

Like their Desktop counterparts, Mobile Fixers and Learners significantly differed in their foraging, as measured by the patch types they navigated from (Analysis of deviance, $\chi^2(4)=28.49$, $p<0.001$) as shown in Fig. 3. On Mobile the largest differences were in the Editor patch type, which was more often used by Learners, and the Package Explorer patch type, which was more often used by Fixers.

Also like their Desktop counterparts, Mobile Fixers vs. Learners also differed significantly in terms of the cue types that influenced their navigations (Analysis of deviance, $\chi^2(6)=19.73$, $p=0.003$) as shown in Fig. 4. Mobile Fixers more often attended to Source Code Content Inspired cue types (53% Fix versus 31% Learn), whereas Mobile Learners more often attended to Domain Text (21% Learn versus 6% Fix) and Output Inspired (7% Learn versus 0% Fix) cue types.

TABLE 2: Mobile participants' navigation patch types. The rightmost column is participants' total number of navigations from that patch type.

Patch type	Information and navigational links	#
Editor	Shows the source code in a file	95
Package Explorer	Provides a list of the files in the project.	47
Search Results	Provides a list of occurrences of user-entered text or a user-selected identifier in an Editor patch. Items can be opened in Editor patch.	64
Stack Trace	Provides a list of code locations on an execution path that produced an exception in the running Vanilla Music program. Items can be opened in the Editor patch.	5
Debug	Provides a list of the currently executing code and program state for a user-specified breakpoint.	6

2) Fixers' Foraging vs. Learners' Foraging

Although the fact that Fixers vs. Learners foraged differently on Desktop generalized to Mobile, the particular patch types they favored did not generalize. Fig. 5 (left) shows commonly navigated patch types for Fixers (top) and Learners

TABLE 3: The cue types Mobile participants followed, and participants' total number of navigations in which they followed each cue type.

Cue type	Participant utterances about...	#
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	67
Level of Abstraction	... cues related to the level of abstraction of a code location they had seen; e.g., "This method is too specific"	40
Domain Text	... cues related to text they had seen specific to Vanilla Music's domain, such as "playlist"	20
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	15
Output-Inspired	... cues related to Vanilla Music output they had seen, such as thrown exceptions (errors) or GUI widget labels	5
File Type	... cues related to the type of a file they had seen, such as Java vs. XML	6
Documentation-inspired	... cues related to external documentation they had seen, such as the bug report	2

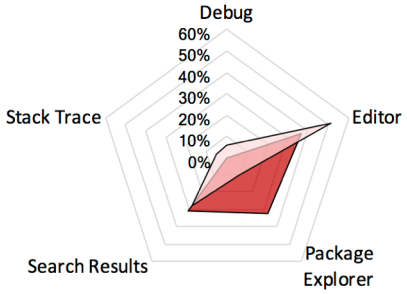

Fig. 3: Mobile Patch Types: The proportion of patch types from which each treatment's Mobile participants made navigations (Fix in dark red, Learn in light red; medium red is where they overlap). We include the 5 most common patch types.

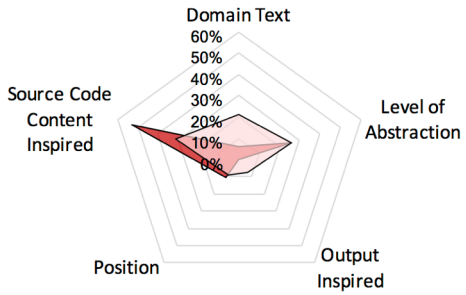

Fig. 4: Mobile Cue Types: The proportion of cue types (5 most common) attended to by Mobile participants for each treatment (Fix in dark red, Learn in light red; medium red is where they overlap).

(bottom) on both environments. The graphs show little overlap. The particular cue types they favored did not generalize either. Fig. 5 (right) shows the five most often attended to cue types for Fixers (top) and Learners (bottom) in both environments, again showing little overlap between the ways in which Fixers or Learners navigated in the two environments.

Note, however, that the main result generalizes: Fixers vs Learners in the Mobile environment still differed significantly from one another, which generalizes on the original study's findings of production bias on Desktop.

B. RQ2 Results: A theory-driven look

Given that Fixers vs. Learners behaved differently on both platforms, why were their differences different across platforms? To answer this question, we consider platform behaviors at a core IFT level.

As explained in Section II, IFT's scent construct—which predators "sniff out" from cues—is defined as predators' attempts to optimize their *expectation* of the value per cost of navigating to the patch pointed to by that cue [38], i.e., E(V)/E(C).

In foraging to individual patches (e.g., individual methods), programmers cannot predict *actual* V/C unless they have been to that patch before. However, a key advantage of our study is that our results are at the granularity of *types* of patches and cues (e.g., the Editor patch type covers all methods viewable in the Editor). At the type level, programmers' expectations of a patch type's value per cost (E(V)/E(C)) will soon become informed by their experiential learning of IDE feature types' *actual* value and cost (V/C). For example, the first time a programmer visits Eclipse's Search Results patch type, she learns that this patch type delivers not only the term she searched for, but also the line number, surrounding text, which file it is in, and where in the project's folder structure that file is located—i.e., the value that patch type always delivers. She also learns that selecting a search result will open the relevant file with the cursor at the line of interest, thus informing her of typical costs associated with using that patch type. As a result, over time E(V)/E(C) will approach V/C, at the level of patch types.

Given this, what we should expect to see is programmers favoring patch types that tend to return the greatest V/C for their goals. And this is precisely what the participants did—the frequency ranking of their visits by type *on both platforms* followed the same order as max(V/C). However, what it took to maximize this ratio differed on Mobile vs. Desktop. On Mobile, the cost C to navigate between patch types tended to be high; thus cost dominated the fraction. (That is, if C in V/C is high, V has little effect on V/C's outcome unless V also is high.)

The Mobile costs in Table 4 show that costs C were indeed very high for navigations to/from anywhere except the Editor—and each increase in cost was accompanied by a drop in popularity of the patch type. Fig. 6 adds a measure of how much more expensive patch types became when running the application was required, and Fig. 7 demonstrates costs in terms of additional information loss.

In contrast to the Mobile platform, on the Desktop platform value V was the dominant factor in maximizing V/C. The reason is that cost C was generally low—much lower than on Mobile—a few clicks/keystroke actions, occasionally a 2-second delay to get the application running, and *no* information loss. (That is, since C in V/C is low, V determines V/C's outcome unless V also is low.) As Table 5 shows,

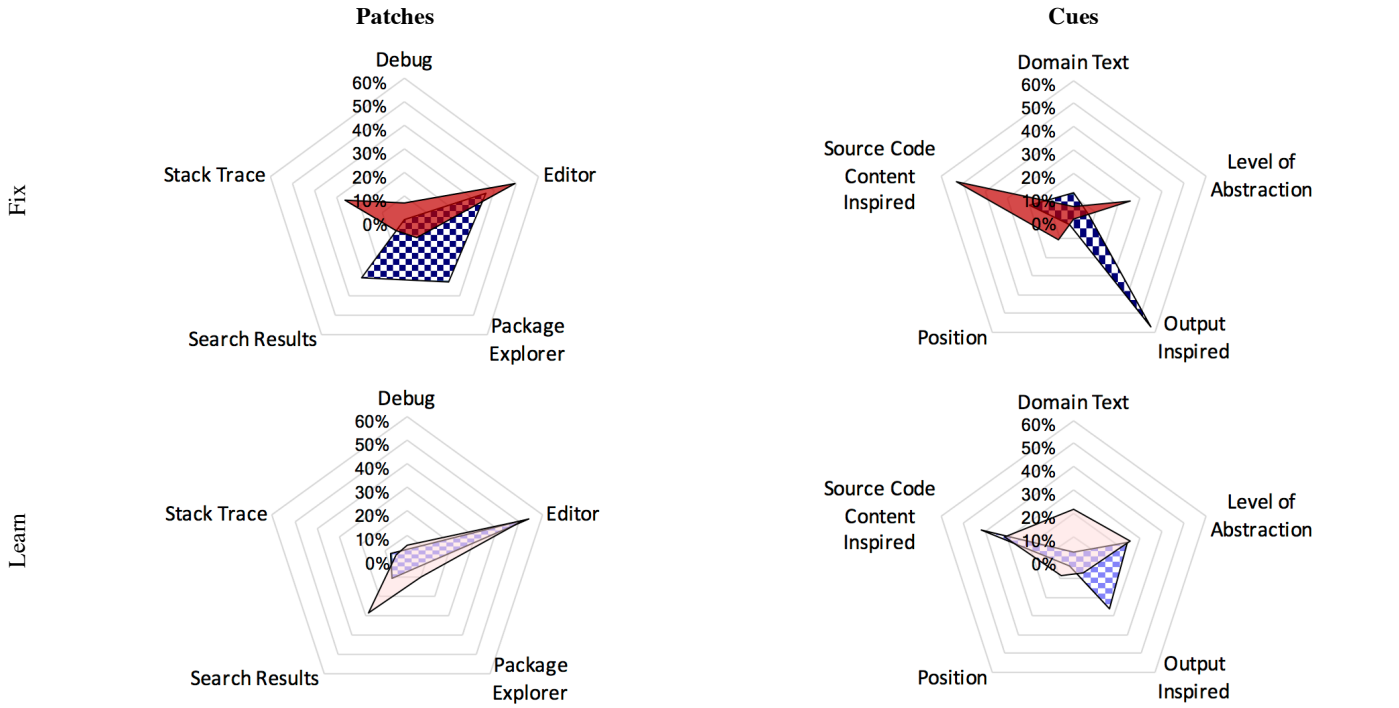

Fig. 5: (left): Patch types for Fixers and Learners. The proportion of patch types from which each environment's Fix (top) and Learn (bottom) participants made navigations (Desktop in blue, Mobile in red, Fix is dark, Learn is light). (right): Fix Cue types for Fixers and Learners. The proportion of cue types attended to by Fix (top) and Learn (bottom) participants for each environment (Desktop in checkered blue, Mobile in solid red, Fix is dark, Learn is light).

Desktop popularity ranks went lockstep with value: "Indispensible" for the Editor, because it is impossible to see and edit a chunk of code any other way; then the only two widely used ways to see dynamic information (e.g., variable values); and finally optional alternative views of static code.

The comparison between Mobile participants' cost-driven behavior with Desktop participants' value-driven behavior suggests the following hypothesis. Mobile costs of some patch types were so high—an order of magnitude higher than Desktop's (Table 4 and Table 5)—that participants simply could not afford some of the information they would have liked if it had been as inexpensively available as on Desktop. This suggests that Mobile IDE tool builders would do well to prioritize reducing the navigation costs of using their IDEs.

C. RQ3 Results: Research Challenges Go Mobile

The Desktop value/cost study identified a "lower bound" on a predator's information foraging cost per value received [36] using information scent. Since scent is defined as the predator's advance predictions of cost per value [39], a lower bound depends on how well a predator can *estimate in advance* of their navigation(s) the actual cost they will pay for the actual value they will receive.

The Desktop value/cost study then identified seven problems participants had in estimating value and cost, thereby interfering with their ability to achieve the lower bound. A literature analysis of over 300 papers [36] showed that these problems remain largely unsolved. In this section we consider whether and how the same problems arose in the Mobile environment.

TABLE 4: MOBILE PATCH-TYPE NAVIGATION COSTS C AS DETERMINERS OF PATCH-TYPE POPULARITY (RANK). REQUIRED ACTION COSTS OF NAVIGATING FROM EDITOR TO OTHER PATCH TYPES IN MOBILE ENVIRONMENT, MEASURED IN UNITS OF TAPS/SWIPES. NAVIGATIONS AWAY FROM EDITOR REQUIRED ERASING PREVIOUS INFORMATION CONTENT DUE TO LIMITED REAL ESTATE (FIG. 7).

	Mobile rank	Mobile costs: actions, time, and information loss
Editor	1	0
Search Results	2	0-3, + lose panel content
Package Explorer	3	0-3, + lose panel content
Debug	4	5+, +run*, + lose panel content+*
Stack Trace	5	6-8, +run*, + lose panel content+*

**running the application. Time cost: Fig. 6. Information loss: entire screen.*

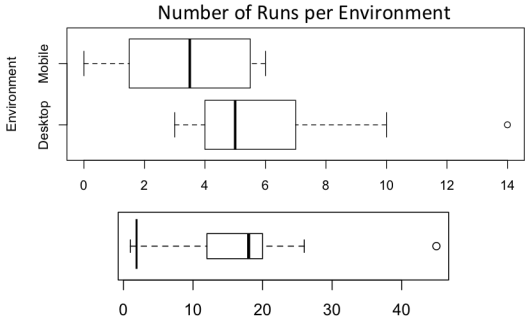

Fig. 6: (Top) Runs: Mobile participants tended to run the application less often than Desktop participants. (Bottom) Time: Getting VanillaMusic Mobile running took significantly longer than getting jEdit Desktop running (Fishers Exact Test, p<.0001). Categories in the Fisher's exact test were runtimes of >2 and <=2 seconds. VanillaMusic Mobile cost to start was 1-45 seconds (median 18 seconds), whereas the maximum jEdit Desktop cost to start was 2 seconds (vertical line at 2-second mark).

1) Value estimation problems

Three of the problems identified in the Desktop value/cost study were value estimation problems, and value estimation problems were also frequent for Mobile participants. Recall (Section III) that after the debugging task, we asked participants whether their expectations of the value each navigation would deliver had been met (Methodology section). For Mobile participants, 50% of responses were negative; in the Desktop value/cost study, the percentage was almost identical: 51% of responses were negative.

The False Advertising Problem: False advertising in IFT [36] is an implied promise by a cue that is not delivered by the patch to which it points. For example, a cue in the form of a method call could imply that the method manipulates a data structure, when in reality the method simply updates the UI or serves as "driver" code whose sole purpose is to call other code to perform all the work. False advertising arose for two of the Mobile participants (Table 6). For example, P5F-M navigated to ShowQueueAdapter.java expecting to find code to add or delete an item, but instead found UI code:

> *P5F-M: <the code> seemed to be more of a UI thing ... <so I> wasn't sure this was where I should be looking.*

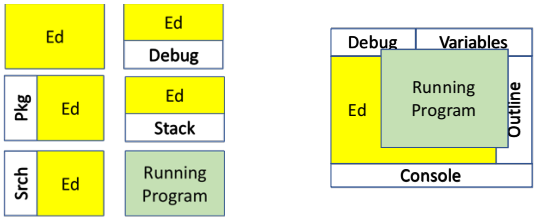

Fig. 7: (Left): *Six mobile screen layouts*. Participants had a maximum of 2 panels open at once, as in the first five depictions. The extra panel could be moved between the left and the bottom. For example, navigating to the Stack Trace would cover, say, the Search portion of the screen. Running the program covered the entire screen, so it was not possible to see the Editor while running the program. (Right): *A desktop screen layout*. Participants could have as many panels and windows open at once as they wanted. The running instance of the program did not need to cover the entire screen, so other patches were still visible.

TABLE 5: DESKTOP PATCH-TYPE VALUE V AS DETERMINERS OF PATCH-TYPE POPULARITY (RANK): INDISPENSABLE FIRST (TO SEE/EDIT CODE), THE ONLY TWO POPULAR WAYS TO GET DYNAMIC INFORMATION SECOND, AND OPTIONAL STATIC VIEWS LAST. NO INFORMATION COSTS WERE INCURRED BY NAVIGATING AMONG PATCH TYPES.

	Desktop rank	Desktop info. value	Desktop costs
Editor	1	Indispensable static view for seeing low-level code details.	0
Debug	2	One of 2 ways to see dynamic info	1-34 + ≤ 2 sec*
Stack Trace	3	The other way to see dynamic info	0-14 + ≤ 2 sec*
Search Results	4 (tie)	Optional view of code	2-5
Package Explorer	4 (tie)	Optional view of structure	0

**running the application cost ≤ 2 seconds (Fig. 6), and 0 information loss.*

Synonym Problems: Many researchers have shown IFT benefits from automatically harvesting word similarities using approaches like TF-IDF and LSI [32, 43, 49]. These benefits come from reducing false negatives, i.e., finding relevant patches even if they do not contain the exact word sought. However, the Desktop study revealed that synonyms also can increase false positives by enlarging the predator's search space with patches that are not relevant. This problem arose for 4 of 8 Mobile participants (Table 6). For example, P2F-M navigated to AudioPickerActivity.java because the words "activity" and "picker" suggested to P2F-M that:

> *P2F-M: <it> might have an enqueue song operation… but it turned out not to be relevant.*

Answering the "wrong" question: This problem is a mismatch between a forager's "question" (goal) vs. the kind of "answer" delivered by an apparently promising cue (e.g., following a method name cue to its method) [36]. For example, the Desktop foragers' questions were of the "where does" variety, such as where in the code a variable actually gets updated. However, many method names are not intended to answer that question. Mobile participants faced this problem too—6 out of 8 did (Table 6). For example, when P5F-M navigated to PlaylistTask.java, he found that it did not answer his "where does" question after all:

> *P5F-M: <trying to figure out> where the interaction is between the playlist and queue…<but PlaylistTask.java> probably had nothing to do with manipulation of the queue… it was <just> some kind of interface.*

2) Cost estimation challenges

The Desktop value/cost study also identified three open research challenges related to cost.

Prey in Pieces: This problem refers to bits and pieces of the prey scattered across multiple patches [36]. For example, Mobile participant P1F-M navigated within PlaylistActivity.performAction(), but realized he would need to also gather information in related code in addSongs() and QueryTask.java (Fig. 8's 1 and 2). Rather than bearing the cost of piecing this all together, he gave up (Fig. 8's 3). This problem arose for 7 of our 8 Mobile participants (Table 6).

The (Seemingly) Endless Path: The endless path problem is a path to the prey so long that the forager gives up before reaching the prey [36]. For example, when:

> *P7F-M: "wanted to see how they set the query <list of songs> in this method",*

TABLE 6: NUMBER OF PARTICIPANTS WHO EXPERIENCED SYMPTOMS OF THE 7 RESEARCH CHALLENGES.

Problems & Research Challenges	# Mobile Participants	# Desktop Participants
Value Estimation Problems:		
#1: False advertising	2/8 (25%)	5/10 (50%)
#2: Synonyms	4/8 (50%)	3/10 (30%)
#3: "Wrong" question	6/8 (75%)	7/10 (70%)
Cost Estimation Problems:		
#4: Prey in pieces	7/8 (88%)	8/10 (80%)
#5: Endless paths	6/8 (75%)	7/10 (70%)
#6: Disjoint topologies	4/8 (50%)	10/10 (100%)
The Scaling Up Problem	Affects all the above	

she navigated from the addSongs() method to the runQuery() method to a search for usages of runQuery to a search result, and believed she had found a useful path. However, in her next navigation she did not find what she expected to find, and she changed course (illustrated in Fig. 9). Six of the 8 Mobile participants experienced this problem (Table 6).

The Disjoint Topologies Problem: When some topologies are not linked to related topologies [36], problems arose for both Desktop and Mobile participants. For Mobile participants, one situation was having pay high costs to map between the running instance of the application and its source code; others were trying to understand relationships between external Android frameworks or the Android manifest file and the Vanilla Music code.

> *P3L-M,* after running the application, *"the bug is actually...clicking on one thing in the library is not the same as the behavior when I do this long press and hit play. And so for this I should find out where the long press play originates,"* which prompted him to look through source code for *"some kind of 'on click' method."*

> *P1F-M:* looked through the manifest file for Android activities *"that might be indicative of a list of songs for playback,"*

Four of the 8 Mobile participants experienced disjoint topology problems (Table 6).

3) The Scaling Up Problem

The final research challenge identified in the Desktop value/cost study was scaling up: finding ways to support foragers' ability to estimate a navigation sequence's costs and ultimate value—*before* paying the cost—for patches that are more than one click away. Although there are a few counter-examples in the literature [36], most research to help foragers

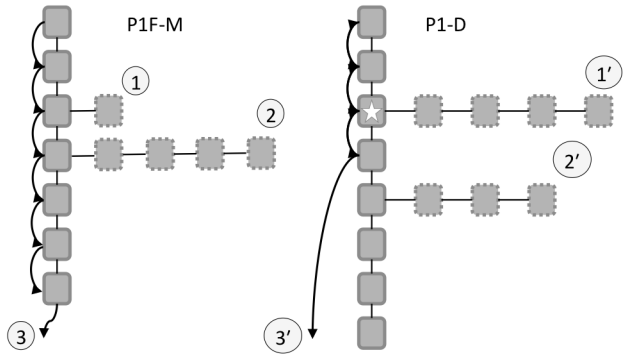

Fig. 8: Prey in Pieces. (Left): Mobile participant's P1F-M navigated to the solid methods, but realized he also needed (1 and (2) at the end of the dotted methods. (Right [36]): Desktop participant P1-D's path looked remarkably similar to P1F-M's. He too needed to piece together information to get the needed information (1' and 2'), but gave up instead (3').

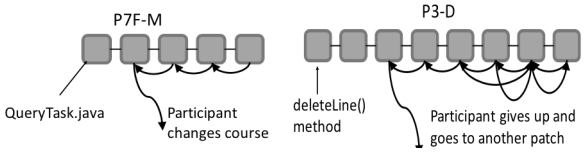

Fig. 9: Endless paths: (Left): P7F-M started down a path, but gave up 1 click before arriving at the right method. (Right [36]): P3-D navigated several frames down the debugger's stack frames looking for methods related to folding or deletion. He gave up 3 clicks short of the right method.

align their estimates of value with real value (and likewise for cost) are akin to tooltips, showing only the nearest neighborhoods but not providing a longer-view "map" of values and costs foragers can expect. An example of the one-click-away sort is a Desktop tooltip showing a method's header.

For Mobile participants, the scaling-up problem exacerbated almost all of the above problems—almost every patch type was inherently more "myopic" (nearsighted) than its Desktop counterpart. For example, Fig. 10 (left) shows how much less information the Package Explorer patch type provided on Mobile than on Desktop. Likewise, the Editor in Mobile is more myopic than Desktop (not shown) because there is no hover text in Mobile. Desktop is able to use hover text to provide a description of code without the programmer needing to navigate away. Even the one patch type in Mobile that did allow a look more than one click away, Search Results, provided less information than its Desktop counterpart because Mobile truncated long lines of code.

Part of the reason the Mobile environment had fewer informational affordances may lie in Mobile IDEs' newness relative to Desktop IDEs, but the problem goes beyond newness. Mobile environments are severely constrained both in terms of screen space and in terms of affordances possible. For example, only Desktops allow hovering and tooltips, multiple mouse buttons, etc. This suggests that solving the Scaling Up problem for Mobile foraging will be even more challenging than solving it for Desktop foraging.

V. Discussion: Replicating from Desktop to Mobile

One thing our study brings out about replicating desktop-based studies to mobile is the tension between consistency with the desktop study conditions vs. authenticity. The fuerther a replication on mobile from the desktop study's environment, task, affordances, etc., the lower its ability to test the generalizability of the desktop results. However, if the replication on mobile is too close to desktop conditions, it loses authenticity as a mobile experience. Addressing this tension between consistency and authenticity matters to our community's ability to bring years of desktop-based research to inform mobile tools.

Our solution to this tension was a two-pronged strategy: (1) we did not sacrifice authenticity (so the IDEs, apps being debugged, and bugs were authentically mobile), and (2) we leveraged IFT to obtain consistency at the level of theory.

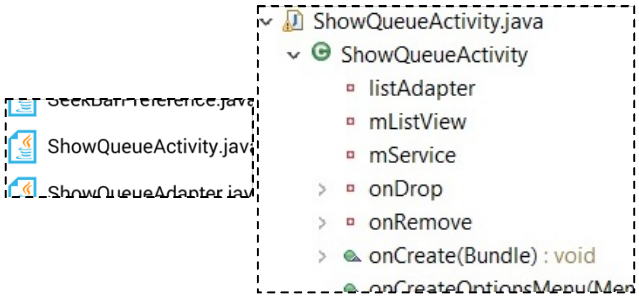

Fig. 10: (Left): The Package Explorer in the Mobile environment lists only the file names. (Right): The Package Explorer in the Desktop environment (shown here on the same code base) allows listing multiple levels of elements in the file.

Leveraging IFT allowed us the ability to abstract beyond *instances* (e.g., particular IDEs, apps, and concrete information content) to the core of IFT—participants' attempts to maximize E(V)/E(C) in both environments. Particularly useful was the notion of patch types, which we leveraged to understand the participants' behaviors in terms of the costs vs. value they expected from each patch type.

The study's IFT foundations also enabled us to generalize participants' foraging difficulties across the two environments and relate them to open research challenges. As Table 6 showed, the extent to which each of these problems arose in the two environments adds to the mounting evidence of the extent to which existing IDEs and software tools subject programmers to onerous and pervasive foraging obstacles.

However, we caution that, to the best of our knowledge, we are the first to attempt replicating from desktop to mobile in this fashion. Additional studies using our replication approach are needed, to gather more evidence of results being due not to differences in the particular UIs, tasks, and code base participants experienced, but rather to theory-based phenomena consistent with those observed in our study.

VI. Conclusion

In this paper we have empirically investigated, through a theory-based dual replication study, whether and how foraging results from a desktop IDE applied to foraging in a mobile IDE. Our results showed several ways in which the foraging results from the desktop-based studies generalized to the mobile environment, even when results about particular affordances and information types did not. In particular:

- *RQ1 (Production Bias meets IFT)*: As with the desktop environment, mobile developers tasked with actually fixing a bug foraged differently than mobile developers tasked with learning enough about the bug to help someone else fix it. This suggests that Minimalist Learning theory's concept of production bias interacts with IFT in similar ways for Mobile as it does for Desktop, an interaction that could be leveraged by IFT-based computational models and tools.
- *RQ2 (How/why)*: The patch types and cue types favored by Desktop participants did not generalize to Mobile. However, from the perspective of maximizing V/C, the behavior results did generalize: on both platforms, participants' usage by type followed the same order as V/C.
- *RQ3 (Problems & challenges)*: Foraging difficulties participants had faced in the Desktop IDE revealed seven open research challenges, and these seven difficulties seem to be approximately as pervasive in the Mobile environment as they were on the Desktop.

Indeed, seeing how this last point played out for our Mobile participants was painful. Still, the environment is arguably one of the best available for mobile development at this time. Although this seems to argue against even trying to perform such a complex information task on a mobile device, the benefits of doing so seem to outweigh the disadvantages in the marketplace, because these kinds of apps are rapidly growing on mobile devices. This suggests an urgency for research that can ultimately address at least some of the research challenges our results reveal.

REFERENCES

[1] A. Agresti. 2003. Categorical Data Analysis. Wiley, 2003.

[2] E. Anderson, S. Li, and T. Xie. 2013. A preliminary field study of game programming on mobile devices. CoRR abs/1310.3308.

[3] J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. ACM CHI, 1589–1598.

[4] S. Burckhardt, M. Fähndrich, P. Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. 2013. It's alive! Continuous feedback in UI programming. ACM PLDI, 95–104.

[5] M. Burnett, J. Atwood, Z. Welch. 1998. Implementing level-4 liveness in declarative visual programming languages. IEEE VL, 126-133.

[6] G. Buscher, E. Cutrell, and M. Morris. 2009. What do you see when you're surfing? Using eye tracking to predict salient regions of web pages. ACM CHI, 21-30.

[7] E. Chi, P. Pirolli, K. Chen, and J. Pitkow. 2001. Using information scent to model user information needs and actions on the web. ACM CHI, 490-497.

[8] C. Choo, B. Detlor, and D. Turnbull. 2000. Web Work: Information Seeking and Knowledge Work on the World Wide Web (vol. 1). Springer Science & Business Media.

[9] C. Crandall and J. Sherman. 2016. On the scientific superiority of conceptual replications for scientific progress. J. Experimental Social Psychology 66: 93-99.

[10] R. Francese, M. Risi, G. Tortora and M. Tucci. 2016. Visual mobile computing for mobile end-users. IEEE Trans. Mobile Computing 15, 4: 1033-1046.

[11] J. Carroll. 1998. Minimalism Beyond the Nurnberg Funnel. MIT Press.

[12] E. Chi, A. Rosien, G. Supattanasiri, A. Williams, C. Royer, C. Chow, E. Robles, B. Dalal, J. Chen, and S. Cousins. 2003. The Bloodhound project: Automating discovery of web usability issues using the infoscentπ simulator. ACM CHI, 505–512.

[13] S. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. 2013. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. ACM TOSEM 22, 2: 14.

[14] W. Fu, P. Pirolli. 2007. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. Human–Computer Interaction.

[15] C. Held, J. Kimmerle, and U. Cress. 2012. Learning by foraging: The impact of individual knowledge and social tags on web navigation processes. Computers in Human Behavior 28, 1: 34-40.

[16] J. Krämer, T. Karrer, J. Kurz, M. Wittenhagen, and J. Borchers. 2013. How tools in IDEs shape developers' navigation behavior. ACM CHI.

[17] S. Kuttal, A. Sarma, and G. Rothermel. 2013. Predator behavior in the wild web world of bugs: An information foraging theory perspective. IEEE VL/HCC, 59–66.

[18] T. LaToza, G. Venolia, R. DeLine. 2006. Maintaining mental models: A study of developer work habits. ACM/IEEE ICS, 492–501.

[19] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. ACM CHI, 1323-1332.

[20] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart. 2010. Reactive information foraging for evolving goals. ACM CHI, 25-34.

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

[22] S. Li, T. Xie, and N. Tillmann. 2013. A comprehensive field study of end-user programming on mobile devices. IEEE VL/HCC, 43–50.

[23] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. 2014. On the comprehension of program comprehension. ACM TOSEM 23, 4: 31.

[24] P. Mayring. 2014. Qualitative Content Analysis: Theoretical Foundation, Basic Procedures and Software Solutions. Beltz, Klagenfurg.

[25] C. Mbogo, E. Blake, and H. Suleman. 2016. Design and use of static scaffolding techniques to support Java programming on a mobile phone. ACM ITiCSE, 314-319.

[26] S. McDermid. 2013. Usable live programming. ACM Onward!, 53-62.

[27] T. Nabi, K. Sweeney, S. Lichlyter, D. Piorkowski, C. Scaffidi, M. Burnett and S. Fleming. 2016. Putting information foraging theory to work: Community-based design patterns for programming tools. IEEE VL/HCC, 129-133.

[28] T. Nguyen, C. Csallner, and N. Tillmann. 2013. GROPG: A graphical on-phone debugger. ACM/IEEE ICSE, 1189–1192.

[29] T. Nguyen, S. Rumee, C. Csallner, and N. Tillmann. 2012. An experiment in developing small mobile phone applications comparing on-phone to off-phone development. Int. Workshop on User Evaluation for Software Engineering Researchers (USER), 9–12.

[30] N. Niu, A. Mahmoud, and G. Bradshaw 2011. Information foraging as a foundation for code navigation. ACM/IEEE ICSE, 816–819.

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

[32] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, R. Bellamy. 2011. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. IEEE VL/HCC.

[33] D. Piorkowski, S. Fleming, I. Kwan, M. Burnett, C. Scaffidi, R. Bellamy, and J. Jordahl. 2013. The whats and hows of programmers' foraging diets. ACM CHI, 3063–3072.

[34] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. 2012. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. ACM CHI, 1471–1480.

[35] D. Piorkowski, S. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Henley, J. Macbeth, C. Hill, and A. Horvath. 2015. To fix or to learn? How production bias affects developers' information foraging during debugging. IEEE ICSME, 11–20.

[36] D. Piorkowski, A. Henley, T. Nabi, S. Fleming, C. Scaffidi, M. Burnett. 2016. Foraging and navigations, fundamentally: Developers' predictions of value and cost. ACM FSE, 97-108.

[37] P. Pirolli and S. Card. 1995. Information foraging in information access environments. ACM CHI, 51–58.

[38] P. Pirolli and W. Fu. 2003. Snif-act: A model of information foraging on the world wide web. In User Modeling 2003, P. Brusilovsky, A. Corbett, and F. de Rosis (eds.). Springer, 45-54.

[39] P. Pirolli and S. Card. 1999. Information foraging. Psychological Review 106, 4: 643.

[40] T. Roehm, R. Tiarks, R. Koschke, W. Maalej. 2012 How do professional developers comprehend software? ACM/IEEE ICSE.

[41] A. Singh, A. Henley, S. Fleming, and M. Luong. 2016. An Empirical Evaluation of Models of Programmer Navigation. IEEE ICSME, 9-19.

[42] J. Spool, C. Perfetti, and D. Brittan. 2004. Designing for the scent of information. User Interface Engineering.

[43] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. 2008. Identifying word relations in software: A comparative study of semantic similarity tools. IEEE ICPC, 123-132.

[44] S. Srinivasa Ragavan, S. Kuttal, C. Hill, A. Sarma, D. Piorkowski, M. Burnett. 2016. Foraging among an overabundance of similar variants. ACM CHI, 3509-3521.

[45] S. Tanimoto. 1990. VIVA: A visual language for image processing. J. Visual Languages Computing 1, 2: 127-139.

[46] N. Tillmann, M. Moskal, J. Halleux, and M. Fahndrich. 2011. TouchDevelop: Programming cloud-connected mobile devices via touchscreen. ACM Onward! 49–60.

[47] N. Tillmann, M. Moskal, J. Halleux, M. Fähndrich, J. Bishop, A. Samuel, and T Xie. 2012. The future of teaching programming is on mobile devices. ACM ITiCSE, 156–161.

[48] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems? ACM CHI, 258-265.

[49] J. Zhou, H. Zhang, and D. Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. ACM/IEEE ICSE, 14-24.