AN ABSTRACT OF THE DISSERTATION OF

David Piorkowski for the degree of Doctor of Philosophy in Computer Science presented on August 10, 2016.

Title: Information Foraging Theory as a Unifying Foundation for Software Engineering Research: Connecting the Dots.

Abstract approved:

____

Margaret M. Burnett

Empirical studies have shown that programmers spend up to one-third of their time navigating through code during debugging. Although researchers have conducted empirical studies to understand programmers' navigation difficulties and developed tools to address those difficulties, the resulting findings tend to be loosely connected to each other. To address this gap, we propose using theory to "connect the dots" between software engineering (SE) research findings. Our theory of choice is Information Foraging Theory (IFT) which explains and predicts how people seek information in an environment. Thus, it is well-suited as a unifying foundation because navigating code is a fundamental aspect of software engineering. In this dissertation, we investigated IFT's suitability as a unifying foundation for SE through a combination of tool building and empirical user studies of programmers debugging. Our contributions show how IFT can help to unify SE research via cross-cutting insights spanning multiple software engineering subdisciplines.

Information Foraging Theory as a Unifying Foundation for Software Engineering Research: Connecting the Dots

by

David Piorkowski

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 10, 2016
Commencement June 2017

Doctor of Philosophy dissertation of David Piorkowski presented on August 10, 2016

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

David Piorkowski, Author

ACKNOWLEDGEMENTS

First and foremost, I'd like to thank my advisor, Margaret Burnett, who had an uncanny knack for challenging me when I needed to be challenged, encouraging me when I needed encouragement and laughing with me on the occasions when everything felt as if it was falling apart. I truly believe that I was one of the lucky ones, who had an advisor who was just as invested (if not more so) in my success as I was. Thank you Margaret.

Thank you also to the members of my committee for agreeing see me through this the seven-year long journey. Your influence resonates in this work and within me. Thank you Chris Scaffidi, Scott Fleming, Anita Sarma, Peri Tarr, and David Kling.

Without external support, much of this research would not have been possible. These internships provided me not only experience, but also daily mentorship, financial support, and necessary equipment (even replacing it when it died mid-internship). I'm especially thankful for the huge amount of flexibility that my mentors gave me to pursue my own interests, often even at the expense of their own. Towards that end, I'd like to especially thank those who guided me through my internships (presented alphabetically by last name). From IBM: Rachel Bellamy, Pietro Ferrara, Bonnie John, Marco Pistoia, and Omer Tripp and from Oracle: Michael van de Vanter. Their guidance and expertise left a mark on me and this work.

I would also like to thank the numerous additional collaborators over the years who given their time, energy and hours of lost sleep to help bring this work to fruition. They are: Irwin Kwan, Chris Bogart, Austin Henley, Tahmid Nabi, Charles Hill, Sruti Srinivasa Ragavan, Amber Horvath, Jamie MacBeth, Liza John, Cal Swart, and Josh Jordahl. It has been a joy to work with all of you.

HCI research is often not possible without willing participants. Our research group thanks our study participants for their time.

Our group would also like to thank IBM for support under an OCR grant and my PhD Fellowship, NSF for support under NSF-1314384 and NSF-1302113, AFOSR for support under FA9550-09-1-0213. The views and conclusions in this dissertation are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of AFOSR, IBM, or the U.S. Government.

Finally, none of this would have possible without the support and sacrifices of my family and loved ones. Thank you Mom, Dad, Jess and Béatrice for always being there and always believing in my potential.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

Chapter 1 Introduction

Navigating through code is an expensive, yet essential aspect of software debugging. In the course of debugging, programmers ask several questions such as where certain features are implemented, why certain paths of code are executed or how different parts of code relate to each other [Sillito et al., 2006]. Answering these questions often requires programmers to seek out information hidden somewhere in the codebase. For some of the questions that programmers ask, navigating through the code to find answers is non-trivial. Prior work [Ko et al., 2006] has shown that programmers can spend 35% of time on average just on the mechanics of navigation.

To address this problem, several tools have been developed to assist programmers in the time-consuming task of navigating through code (e.g., [DeLine et al., 2005b; Henley & Fleming, 2014; Karrer et al., 2011; Kevic et al., 2014; Krämer et al., 2013; Majid & Robillard, 2005; Singer et al., 2005]). However, there is little foundational knowledge to support the design and evaluation of such tools. Therefore, the generalizability of those solutions tends to be limited. To truly understand programmers' information-seeking behavior, we need theory.

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

As Shaw eloquently explained, scientific theory lets technological development pass limits previously imposed by relying on intuition and experience [Shaw, 1990]. For example, her summary of civil engineering history points out that structures (buildings, bridges, tunnels, canals) had been built for centuries—*but only by master craftsmen*. Not until scientists developed theories of statics and strength of materials could the composition of forces and bending be tamed. These theories made possible civil engineering accomplishments that were simply not possible before, such as the routine design of skyscrapers by ordinary engineers and architects [Shaw, 1990]. And indeed, in computer science, we have seen the same phenomenon. For example, expert programmers once built compilers using only their intuitions and experiences, but the advent of formal language theory brought tasks like parser and compiler writing to the level that undergraduate computer science students now routinely build them in their coursework [Aho et al., 1986].

Our choice of theory to understand programmers' information seeking is Pirolli's Information Foraging Theory (IFT) [Pirolli & Card, 1999; Pirolli, 2007]. IFT provides constructs and propositions to explain and predict how people seek information. It has been well-validated over several domains such as document collections [Pirolli, 1998; Pirolli & Card, 1995, 1998], web pages [Chi et al., 2000, 2001; Fu & Pirolli, 2007; Pirolli, 2005, 2007; Pirolli et al., 2005; Pirolli & Fu, 2003] and program code [Lawrance et al., 2007, 2008a, 2008b, 2010, 2013; Piorkowski, 2013; Piorkowski et al., 2011].

We would like achieve a higher level of connection among works to support information seeking by leveraging IFT as an abstraction. Therefore, in this thesis, we aim to "connect the dots" between established SE literature and IFT. We focus our efforts on factors that affect programmers' navigation in code because navigation is fundamental to many software engineering activities. We address the following research question:

Thesis Research Statement

> *Information foraging theory can help to unify SE research via cross-cutting insights spanning multiple software engineering subdisciplines.*

Chapter 2 Background and Related Work

2.1 Information Foraging Theory Constructs and Propositions

Originally derived from Optimal Foraging Theory [Stephens & Krebs, 1986], which describes how animals seek food given a specific environment, Information Foraging Theory [Pirolli & Card, 1999] adapts its constructs and propositions to explain and predict how people seek information. A person, the *predator*, seeks information in an *information environment*. The environment is a collection of *information patches* (documents, methods in code) connected by *links* (e.g., hyperlinks, menu items, scrolling) to form a *topology*. Each link has a cost of traversal (e.g., the time it takes to traverse the link, where time is influenced by a person's physical and cognitive speed). Predators navigate the topology by moving between patches by following an outgoing link from their current patch or backtracking. Figure 2.1 depicts an example topology.

A patch is a collection of *information features* (words, syntax, pictures, etc.), which contain information that a predator can process. A predator processes patches in order to find their *information goal* (the set of information the predator is seeking). The information features that match a predator's goal are termed *prey*. Like links, each information feature has a processing cost associated with it based on the time to read and understand the feature. Some of these information features are associated with an outgoing link, and are called *cues.* Cues act as signposts for the predator and give a clue as to what may be on the other end of the link that a cue is associated with. Figure 2.2 depicts two example patches.

According to IFT, at each patch, the predator is faced with three choices: forage within the patch, move to another patch, or enrich the environment. In the first case, called *within-patch foraging*, the predator processes some of the information features within the patch looking for information related to their current information goal. In the second case, *between-patch foraging*, the predator decides to traverse a link and move to another, more promising patch. In the final case, called *enrichment*, the predator changes the environment. Enrichment can take many forms such as highlighting a word or adding a patch (creating a set of search results) or a link (adding a bookmark). The features provided by the information environment limit the types of enrichment available to the predator.

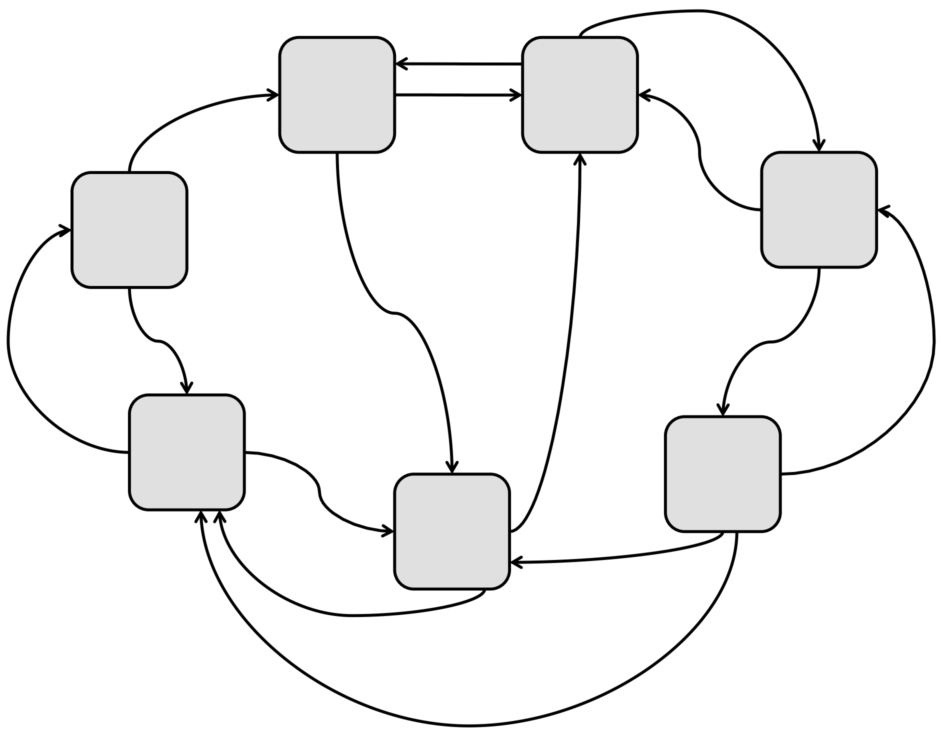


Figure 2.1. Example topology. Each square represents an information patch. Each directed edge is a navigable link from one patch to another. Each link's weight represents the cost of traversing the link [Pirolli, 2007].

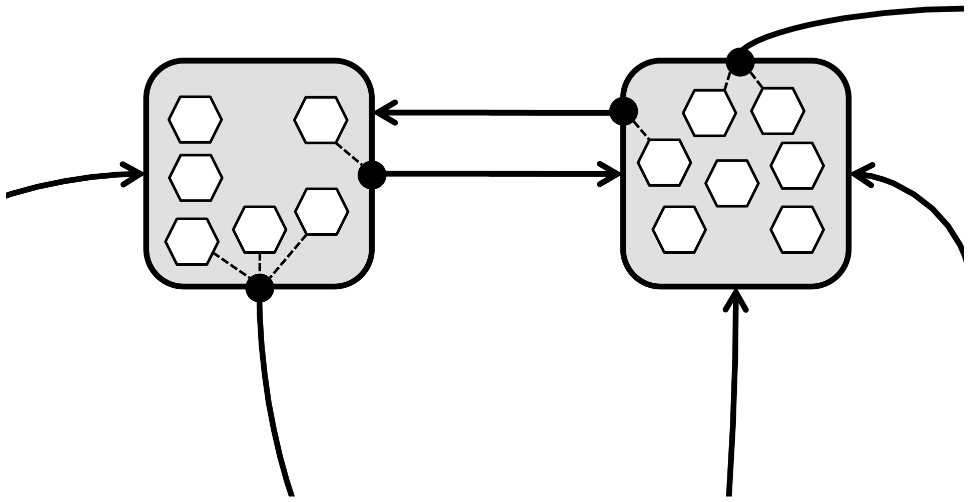


Figure 2.2. Example of two information patches. Each patch contains a set of information features, depicted as hexagons. Some information features are cues, which are associated with outgoing links. Dashed lines

indicate these associations. Each hexagon is also annotated with a processing cost, shown in each hexagon [Pirolli, 2007].

IFT's key prediction is that the predator tries to maximize the amount of information gained per cost of interaction, which Pirolli formalized as follows:

$$\text{Predator's desired choice} = max\frac{V}{C}$$

where V is the value of the information gained and C is the cost of the interaction, that is, the cost of processing a patch, the cost of traversing a link or the cost of enriching the environment.

Generally, predators do not have perfect information about the amount of information gained or the cost of interaction, so they make their choices based on their estimations of value and cost, which Pirolli formalized as follows:

$$\text{Predator's selected choice} = max\frac{E(V)}{E(C)}$$

where $E(V)$ is the expected value of the information gained and $E(C)$ is the expected cost of interaction. The difference between the predator's desired choice and the predator's selected choice is based on how accurately the predator estimated V and C.

To evaluate where to go next, the predator follows links with the highest *information scent*. Information scent lives in the predator's head, and is summarized by Pirolli as "terse representations of content … whose trail leads to information of interest" [Pirolli, 2007]. In other words, information scent captures how the predators process information features and cues in order to make their next foraging decision. Predators seek to maximize the expected value of information gained while minimizing the cost of traversing the link. Formally, in the case of between-patch foraging, the predator is currently seeking current prey G and has a set of available outgoing links L, available to choose from the current patch. Each link $l \in L$, has a set of associated cues J_l (such as words in the hyperlink). IFT predicts that predators will tend to evaluate links according to the following function:

$$\frac{E(V)}{E(C)} = \sum_{i \in G} \sum_{j \in J_l} W_j S_{ji}$$

where S_{ji} (information scent), is the predator's estimation of the likelihood that some prey $i \in G$ is at the other end of the link given the presence of some cue $j \in J_l$. W_j represents the amount of attention that a predator gives to a particular cue j. Both S_{ji} and W_j are non-negative numbers in the above equation.

2.2 Initial application of IFT: Document Collections

The formative work on IFT was developed in the domain of large document collections. In their first papers on IFT, Pirolli and Card introduced the theory and investigated its application in two document foraging tasks: collecting relevant documents in large document collection using the Scatter/Gather interface [Pirolli & Card, 1995] and finding relevant research publications by following the publications' citation link structure using Butterfly [Pirolli & Card, 1998]. In both these works Pirolli and Card operationalized each of IFT's constructs first to the domain and then developed a model to explain users' foraging behavior. In the case of [Pirolli & Card, 1995], Pirolli and Card developed a dynamic programming model and compared it to users' actual behavior. In [Pirolli & Card, 1998], they extended an existing model of cognition, ACT-R [Anderson, 1993], with IFT-derived predictions and found their cognitive model also matched human users' foraging. In follow up work, Pirolli demonstrated how a dynamic programming model for Scatter/Gather could be used to analyze and test improvements to Scatter/Gather's interface [Pirolli, 1998].

2.3 IFT and Web Foraging

One particularly rich application of IFT was in the web domain. The first application of IFT in this space was a method for collecting web pages through the use of a model that combined structural (links on web pages), lexical (word similarity) information with spreading activation [Anderson & Pirolli, 1984] to group web pages into functional categories [Pirolli et al., 1996]. These methods would be repurposed to explain and predict how users forage on the web [Chi et al., 2000, 2001; Fu & Pirolli, 2007; Pirolli, 2005, 2005; Pirolli & Fu, 2003] and later applied to tools that identified usability issues with web sites [Card et al., 2001; Chi et al., 2003; Pirolli et al., 2005] or informed their design [Nielsen, 2003; Spool et al., 2004]. IFT has also been leveraged in tools to help users collect and structure information during web foraging [Kittur et al., 2013], find relevant links on a web page [Olston & Chi, 2003], and assess the difficulty of a web foraging task [John et al., 2013].

The models that were used to explain users' foraging behavior on the web were originally derived from techniques developed in cognitive psychology to explain human decision making processes. Models such as the ones used in SNIF-ACT [Pirolli & Fu, 2003] and WUFIS [Chi et al., 2003] are derived from a spreading activation cognitive architecture first introduced by Anderson and Pirolli in [Anderson & Pirolli, 1984]. In this architecture, a human's long-term memory is represented as a graph of nodes. Each node contains a chunk of information with more recently accessed nodes having a higher activation (value on the node). Each edge represents an association between two chunks of information where more-similar items have a higher association. Given an information goal, these models spread activation through the graph to model how human memory works. Nodes with the highest values after activation represent the relevant information chunks for the given goal. This spreading activation approach is used to model information scent, and is the key behind predicting user navigation on the web.

In the case of SNIF-ACT, Pirolli and Fu's cognitive model was based on a set of production rules similar to ACT-R [Anderson, 1993]. The SNIF-ACT cognitive model represented each web page of a site as a patch and calculated the scent of each link based on lexical similarity between text surrounding the link and the goal. The model would progress through several states representing a web user's cognitive processes. When the model reached a state where it could choose a link, the model would either click a link or leave the page depending on what scent values were calculated. The model's behavior of selecting links and backtracking closely approximated how humans foraged on web sites.

WUFIS (Web User Flow by Information Scent) combined ideas from SNIF-ACT and Information Retrieval to predict web site navigation and to identify which pages on a site were difficult to find. WUFIS would scrape a website and construct a graph representing the site. Individual pages were represented as vertices and the edges represented the page-to-page links between pages. Once an information goal was provided, each page's links would have their scent calculated. Then, the graph's edge values were updated to represent the probability that the corresponding link would be taken. With the graph complete, WUFIS performed a user-flow simulation that had a large number of users navigate through the site following edges according to their probability. The result of the simulation provides a measure of which pages users visited most often for a given information goal. WUFIS served as the backbone behind the Bloodhound project's usability analysis [Chi et al., 2003].

One of the first attempts at understanding foraging with multiple users was Collaborative Foraging, a combination of collaborative filtering and IFT [Schultze, 2002]. Schultz developed a tool called WebWaggle, which allowed users to create and organize list of bookmarks to web pages. As lists are created and shared, the system makes IFT-based recommendations for other pages a user may be interested in based on the pages that other users have added and categorized. Chi and Pirolli analyzed the web site bookmarks of several users to determine similarities and differences in bookmarked pages across several topics to help design a collaborative search tool [Chi & Pirolli, 2006]. They found that users had more web sites in common than expected and that greater diversity among users leads to better information foraging. Pirolli then extended both IFT and sensemaking, and developed formal models for Social IFT [Pirolli, 2008, 2009]. Others have built on this research to describe how programmers collect and aggregate information from multiple web sites [Evans & Card, 2008a], to provide web search results that are ranked based on users' actions on the web [Luca et al., 2009] or to frame the practices around asking questions in social environments on the web [Evans et al., 2010].

Social Information Foraging Theory extends IFT by explaining how multiple foragers (instead of a single forager) work together when foraging. Social IFT differs from IFT in terms of how the cost and benefit (a synonym for value) are evaluated for each information patch. In IFT, the individual determines the costs and benefit whereas in Social IFT, the group determines the cost and benefit. In this model, *individual* predators make their foraging decisions using the *group's* assessments of cost and benefit.

2.4 IFT and Software Engineering

Information foraging theory has also been applied to explain how programmers forage through code during software maintenance. The first work combining software maintenance and IFT investigated how the constructs of patches, scent and diet could be applied to software maintenance tasks [Lawrance et al., 2007]. In their study Lawrance et al. found that when given the same tasks, programmers tended to visit the same classes and that the lexical content of those classes were similar to the bug report or feature request from the task. Lawrance et al. also investigated what parts of a task's bug reports and feature requests were predictors for classes relevant to the task using lexical similarity [Lawrance et al., 2008a]. They found that for feature requests titles were more accurate than the entire feature request and that for bug reports, the entire content of the bug report was more accurate than the title alone.

These two works led to the development of the first predictive model of programmer foraging behavior, dubbed PFIS, which stands for Programmer Flow by Information Scent [Lawrance et al., 2008b]. PFIS adapted the approach of WUFIS [Chi et al., 2003] of building a graph of the source code using the classes and methods as vertices and the links (structural relationships, method calls, constructors, etc.) between them as edges. The weights of the edges represented the lexical similarity between a bug report or feature request to the text surrounding each link. Activation was then spread along the edges until node values settled. The resulting values represented the probability a given location in code was likely to be visited for a given bug report or feature request. Lawrance et al. then compared PFIS's prediction against participant asked to complete a bug fixing task or a feature addition task. The study's results showed that the PFIS model's predictions closely mirrored that of participants' aggregate navigations, with better accuracy for the bug fix task than the feature addition task, suggesting that bug reports had higher scent than feature requests.

The high predictive accuracy of PFIS suggested that the combination of scent and topology is sufficient for predicting programmer navigation. Yet earlier work described debugging mainly as a process of forming and evaluating hypotheses such as that of Brooks' Theory of Comprehension [Brooks, 1983] or Letovsky's Cognitive Processes of Program Comprehension [Letovsky, 1987]. In [Lawrance et al., 2013], Lawrance et al. investigated the relationship between hypotheses and scent and if hypotheses should be incorporated in models of programmer navigation. Lawrance et al. found (1) that some hypotheses closely parallel scent and are therefore accounted for in IFT and (2) that programmers spent comparatively little time processing hypotheses as opposed to processing scent.

In the same paper, additional analyses focused on reasons why PFIS's IFT-based model was successful. One finding showed that scent seeking occurred throughout the debugging sessions, suggesting that a scent-based approach was appropriate regardless of whether the programmer was looking for a place to start debugging or whether they were fixing the defect. Another finding showed that scent seeking was most often triggered when programmers were looking at source code (as opposed to other parts of the IDE, the executing program or other artifacts external to the IDE but related to the defect), suggesting that source code may be sufficient for representing the topology if other artifacts were not available to a tool or model. A lexical similarity analysis found that the bug report text was a better predictor of navigation than programmers' hypotheses. Taken together, these results shed light on why PFIS accurately predicted programmer navigation.

In the follow-up to PFIS, fittingly called PFIS2 [Lawrance et al., 2010], Lawrance et al. extended the predictive model to be reactive both to the programmer's goals and to changes to the source code. Unlike PFIS, which used the entire codebase to construct its underlying topology, PFIS2 builds and changes the graph as a programmer explores and changes the code. The evolving topology was intended as a more accurate representation of a programmer's ever-changing goals. Additionally, PFIS2 accounted for the programmer's navigational history when making predictions; more recently visited locations were favored over less recently visited locations. The incorporation of evolving topology and programmer history led to a successful predictive model. The results from a seven-month field study, where PFIS2 was deployed to two professional programmers, showed that of the 4,795 predictions made by PFIS2, more than half were in the model's top three predictions. 27% of the navigations were predicted by PFIS2's top choice. The results provided evidence for including programmer's navigation history and an evolving topology when predicting navigations.

To investigate exactly what factors mattered in a predictive model of programmer navigation, we investigated each of the factors within PFIS2, other predictive models such as Parnin and Gorg's [Parnin & Gorg, 2006], and tools designed to support navigation [Cubranic & Murphy, 2003; DeLine et al., 2005b; Robillard & Murphy, 2003; Schummer, 2001; Singer et al., 2005, p. 200; Sinha et al., 2006; Storey et al., 2008; Zimmermann et al., 2005]. With these factors, we developed and evaluated several single-factor and multi-factor models finding that navigation history (Recency in the paper) was indeed an important factor for accurate predictions [Piorkowski et al., 2011]. In my thesis [Piorkowski, 2013], I extended this work and investigated the role of lexical similarity by evaluating lexical similarity at the granularity of both methods and classes. (Although PFIS2 compared predictive accuracy between classes and methods, it did not consider lexical similarity by itself but combined with other factors.) The results from my thesis suggested that in addition to history, lexical similarity was also relevant for predicting programmer navigation.

Besides predictive models, researchers have applied IFT or IFT constructs to several software tools. Niu et al. have investigated how programmer navigation can be used to build a better-informed patch model for code navigation tools [Niu et al., 2011]. Niu et al. also developed an optimal foraging model based on IFT for a requirements tracing tool, identifying opportunities for improvements based on where human analysts differed from the optimal model's predictions [Niu et al., 2013]. Our group built a recommendation system for programmers that utilized IFT to generate its predictions [Piorkowski et al., 2012] (Chapter 3). Inspired by our work on single-factor models, Kramer et al. demonstrated how including navigation options based on a program's call graph improved task completion time [Krämer et al., 2013]. Kuttal et al. have framed how end-users debug web mashups through an IFT lens, identifying cues in that environment, users' debugging strategies and finding implications for design for these sorts of tools [Kuttal et al., 2013]. Our group has also viewed existing software maintenance tools through an IFT lens to explain why those tools were successful [Fleming et al., 2013]. This work identified several design patterns based on IFT that can be used to develop or understand future tools for software maintenance.

Chapter 3 Predictive Factors as Scent

We begin our investigation of how to unify software engineering research with one of IFT's key constructs; *scent*, which explains how predators evaluate cues in a patch when making their next foraging choice. In this chapter, we start with the assumption that the factors underlying several existing papers about tools that support programmer navigation are different types of scent. If the factors are indeed different types of scent, we next determine which of the factors are more important for scent in a tool context. By doing so, we demonstrate the utility in unifying different tools under the same construct because once the tools can be explained under the same abstraction, their individual results can be compared against each other. At the end of the chapter, we will revisit the assumption, drawing on what we learned in the chapter to decide if the assumption is reasonable.

To investigate this question, we first have to choose which navigation factors to investigate. We drew candidate factors based on the results of my Master's Thesis [Piorkowski, 2013], where I evaluated several predictive factors of programmer navigation. These factors were drawn from existing tools that ease navigation by providing shortcuts to code and included factors such as how recently a programmer visited a location in code [DeLine et al., 2005b; Robillard & Murphy, 2003; Singer et al., 2005], what code was modified at the same time [Storey et al., 2008; Zimmermann et al., 2005], textual similarity, method-invocations, and code structure relationships [Cubranic & Murphy, 2003; Schummer, 2001; Sinha et al., 2006]. RQ1 already investigates the most accurate factor Recency, but [Piorkowski et al., 2011] showed that when factors were combined, accuracy improved. Models that included factors based on source code structure and word similarity improved predictive accuracy over Recency alone. Given these findings, we chose to evaluate the factors of recency, code structure and word similarity.

We structure our investigation around the following questions.

First, although Recency was the accurate predictive factor, the question of how much of a programmer's recent navigation history to consider for recommendations remains. As noted in Lawrance's work on Reactive IFT [Lawrance et al., 2010], a programmer often investigates code related to a certain goal for a while—which we refer to here as "building up foraging momentum" related to that goal—before shifting to a new goal. Reactive IFT posits that programmers' current momentum, reflected by recent navigation history, can be used to infer the goal, but it is unclear how many recent navigations to use for this analysis.

- RQ1: How do a Reactive IFT-based tool's assumptions about foraging momentum affect its ability to infer a programmer's goal and produce useful recommendations?

The second question is what other factors to consider. As mentioned earlier, we limited our investigation to the most common factors drawn from the SE literature.

- RQ2: Does a Reactive IFT-based tool that considers word similarity and code-structure cues yield more useful recommendations than another tool that considers only word cues?

Third, because the navigational behavior of programmers changes over the course of a task (as noted earlier), will different operationalizations be more or less appropriate at different points in a task?

- RQ3: Does a Reactive IFT-based tool's ability to provide useful recommendations change as a task progresses?

Fourth, and related to this point, different programmers might obtain different value from Reactive IFT-based tools at different points in a task, raising the question of when and why recommendations are useful.

- RQ4: When and why do programmers find Reactive IFT-based tools' recommendations useful (or not)?

3.1 Related Work

Code navigation is a key part in a learning process whereby the programmer finds the information needed to work out a concrete goal to complete the task at hand. At the start of a maintenance task in laboratory and field studies, a programmer usually searched for code that could serve as an initial focus point [Sillito et al., 2006; Wiedenbeck & Evans, 1986]. Over the course of a task, the programmer typically began to ask questions about how pieces of code were related [DeLine et al., 2005b; Sillito et al., 2006]. Programmers also frequently navigated between locations in code, revisiting places to learn about structural relationships [Parnin & Gorg, 2006; Sillito et al., 2005]. As programmers discovered answers to their questions, they broadened their focus to include more code [Sillito et al., 2005]. Eventually, they planned specific changes that they believed would eliminate a defect or create a new feature [Ko et al., 2006; Sillito et al., 2005].

Various tools are aimed at easing these navigations by providing shortcuts to recommended places in the code that might hold valuable information. Several tools provide "history" links back to code that the programmer has recently visited [Kersten & Murphy, 2005; Parnin & Gorg, 2006]. Others offer shortcuts to places that other programmers historically have read and/or edited after the current location [DeLine et al., 2005b; Hill et al., 1992]. Still other tools provide shortcuts to code or other artifacts based on textual similarity, textual proximity, method-invocation, or nesting [Cubranic & Murphy, 2003; Jakobsen & Hornbæk, 2006; Sinha et al., 2006; Storey et al., 2002].

A limitation of these tools is that they do not explicitly take into account the evolution in goals that typically occurs as a programmer learns during a task. For example, upon visiting a Java method's code at the start of debugging, a programmer might have no idea what needs to be edited; upon revisiting the location later, the programmer might have formulated a goal to make certain kinds of edits. Yet most of the tools above would provide exactly the same shortcuts in both situations, regardless of the programmer's new goal. Of the above tools, only those that show a "history" of recent places would behave differently during a particular task: they would stop showing links to locations that were not recently visited, which is an implicit model of evolving goals at best.

Reactive IFT [Lawrance et al., 2013] beings to address of some these limitations. The purpose of this study is to investigate how to apply this theory to help programmers find the information they need during tasks.

3.2 Methodology

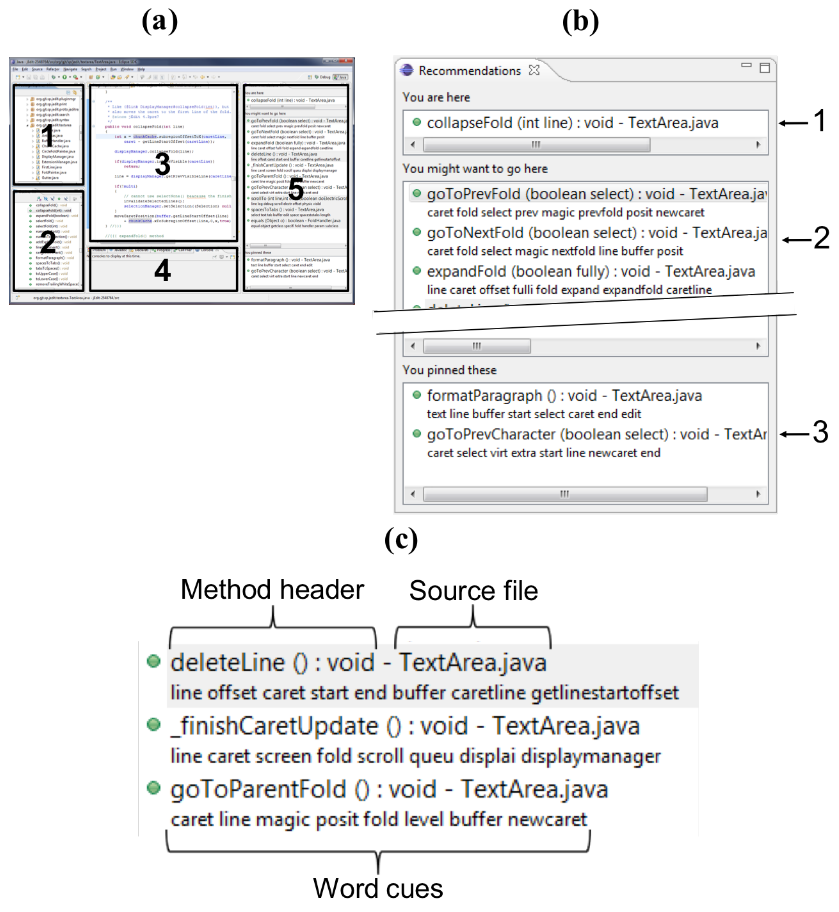
In our empirical study, we invited professional programmers to complete a debugging task using a new Eclipse plug-in tool that supplied links to places in the code that might provide information needed for the task. Within this plug-in, we activated different recommendation algorithms based on different operationalizations of momentum and scent. Specifically, we considered different algorithms in a 2×2 factorial design (with one factor for the operationalization of momentum and another factor for the operationalization of scent). We then measured what proportion of the time participants went to locations recommended by different algorithms. In our analysis, we also examined whether algorithms' recommendations were more or less useful at different periods of the task, and we qualitatively analyzed what kinds of benefits participants obtained.

3.3 Study Environment

We implemented our recommendation system as a plug-in for the Eclipse IDE (Figure 3.1). Interface elements 1–4 in Figure 3.1.a are all those that commonly appear in the Java perspective of Eclipse: (1) Package Explorer view, (2) Outline view, (3) Java editor, and (4) Console view. Interface element 5 is our Recommendation view.

Figure 3.1.b depicts a close-up of this Recommendation view, which had three areas: (1) the current method (i.e., the one that the text cursor last entered), (2) the current recommendations, and (3) methods bookmarked, or *pinned*, by the programmer. Each time the programmer navigated to a method, the current method updated to reflect the navigation, and the recommendations were recalculated (using whichever of our algorithms was activated at the time). The programmer could also manually drag the current method or any recommendation into the pinned area to save it for later.

Each recommendation displayed words to help participants assess its relevance (Figure 3.1.c). These words were sorted based on their importance according to the amount of weight given to them internally by the recommender algorithm active at the time. The Recommendation view also distinguished recommendations to methods the participant had previously visited from recommendations to methods not already visited by highlighting the latter in gray.



Recommendations

You are here

collapseFold (int line) : void - TextArea.java

You might want to go here

goToPrevFold (boolean select) : void - TextArea.ja
caret fold select prev magic prevfold posit newcaret
goToNextFold (boolean select) : void - TextArea.ja
caret fold select magic nextfold line buffer posit
expandFold (boolean fully) : void - TextArea.java
line caret offset fulli fold expand expandfold caretline

You pinned these

formatParagraph () : void - TextArea.java
text line buffer start select caret end edit
goToPrevCharacter (boolean select) : void - TextAr
caret select virt extra start line newcaret end

deleteLine () : void - TextArea.java
line offset caret start end buffer caretline getlinestartoffset
_finishCaretUpdate () : void - TextArea.java
line caret screen fold scroll queu displai displaymanager
goToParentFold () : void - TextArea.java
caret line magic posit fold level buffer newcaret

Figure 3.1. Eclipse interface with our Recommendation view: (a) Eclipse layout, (b) Recommendation view, and (c) examples of three recommendations.

For data collection, our plug-in recorded a log of participant interactions with Eclipse. Additionally, we video-recorded each session and automatically logged screen captures.

3.3.1 Recommendation Algorithms

To investigate whether operationalizing scent as words + structure would produce better recommendations than words alone (RQ2), we implemented one recommender algorithm based on words + structure, PFIS-R, and another based on words alone, TFIDF-R. The PFIS-R recommender is based on the PFIS3 predictive model [Piorkowski et al., 2011], motivated by PFIS3's success in predicting programmer navigation. The other algorithm, TFIDF-R, is based on a vector space model commonly used in information retrieval [Baeza-Yates & Ribeiro-Neto, 1999]. These two algorithms represent the main camps of how to model information foraging: strong reliance on words (e.g., TFIDF, LSA) and balancing words with information structure (e.g., [Chi et al., 2001; Lawrance et al., 2008b, 2010, 2013; Olston & Chi, 2003]). Both share a reliance on words, because word-based approaches have dominated IFT (e.g., [Card et al., 2001; Chi et al., 2003; Fu & Pirolli, 2007; Pirolli, 2007]) and the literature contains some evidence that words can be used to predict where programmers will navigate [Lawrance et al., 2008b]. RQ1 required a need to manipulate the operationalization of the *foraging momentum*—specifically, how much history to use for making recommendations. To take momentum into account, we parameterized each algorithm with δ, the number of navigations to look back in making a recommendation. An algorithm with $\delta=1$ uses only the contents of the last method visited and thus ignores any momentum that the programmer might have built up; we refer to this algorithm configuration as *low momentum*. In contrast, an algorithm with $\delta=10$ uses the contents of the last 10 methods visited and will thus be influenced by momentum built up during those navigations; we refer to this algorithm configuration as *high momentum*. Thus, the greater the value of δ, the more momentum the recommender assumes in making its recommendations. Combining two momentum configurations, $\delta=1$ and $\delta=10$, with the two different algorithms (PFIS-R vs. TFIDF-R) gives the 4 possible combinations of our 2×2 factor design.

The algorithms take as an input the sequence of methods to which a programmer has navigated so far. In the current study, we recorded a navigation to a method m each time the programmer performed an action that caused the text cursor to enter the text of m. For our purposes, the text of a method comprises the method's signature, body, and (Javadoc) comments. Both PFIS-R and TFIDF-R normalize this textual input by excluding punctuation and stop words, which include common words, like "the", as well as the list of Java keywords. Camel-case words are broken into separate case-insensitive words, although the original camel-case word is retained as well (e.g., *setFoldText* would be treated as an input of *setfoldtext*, *set*, *fold*, and *text*). For the remainder of the chapter, *method text* refers to normalized method text.

The algorithms produce 10 recommendations as output, divided evenly between methods previously visited by the programmer and methods not previously visited. Programmers commonly revisit methods to recover mental state and explore new methods to find and understand the program's essential elements [Parnin & Gorg, 2006]. Thus, by recommending both visited and unvisited methods, we sought to support both types of navigation. If there are fewer than 5 previously visited recommendations, the algorithms fill out the remainder of the 10 recommendations with new recommendations, and vice versa. The algorithms would sometimes need to resolve ties when generating this list. The reason is that although the algorithm ranks methods by computing a continuous measure of relatedness (as discussed in the algorithm details below), the variables in those calculations take on a small number of distinct values in practice. When selecting the 10 methods to recommend, ties were resolved by choosing non-deterministically among the tied methods.

PFIS-R Recommender

Figure 3.2 summarizes the PFIS-R recommendation algorithm, which builds on the earlier PFIS2 and PFIS3 models by informing its recommendations using word cues and code-structure cues (i.e., call dependences) in methods that the programmer previously visited. In brief, PFIS-R maintains a graph of word and method vertices such that edges between method vertices capture the structural relationships between methods, and edges between words and methods capture lexical relationships between methods. The algorithm *spreads activation* [Anderson & Pirolli, 1984] over this graph, starting from the vertices for the last δ methods that the programmer visited. To account for momentum, vertices for methods that the programmer visited more recently are initialized with greater activation. The algorithm considers methods with the highest resulting activation to be the best recommendations, and those are the ones that sort first in Figure 3.1.c.

TF-IDF Recommender

In contrast to PFIS-R, TFIDF-R bases its recommendations on lexical similarity only. TFIDF-R treats the code base as a corpus of documents with the (normalized) text of

PFIS-R Algorithm

Definitions:

- Method set M: set of all methods in the code base.
- Navigation history H: sequence of methods to which the programmer has navigated so far.
- Word set W: set of all words in all method text in M.
- Graph $G = (V_m \cup V_w, E_m \cup E_w)$ such that V_m and V_w have a one-to-one relationship with M and W, respectively. For all m_a, m_b in M and their associated method vertices mv_a, mv_b in V_m, E_m contains one and only one edge connecting mv_a and mv_b if and only if the body of m_a contains a call to m_b or the body of m_b contains a call to m_a. For each word w in W and method m in M, E_w contains one and only one edge connecting wv and mv if and only if w is in the method text of m.

Steps for making recommendations:

- Set activation of each vertex in G to 0.
- For each method m such that m is the kth method in H and $|H|-\delta < k$, increment the method vertex mv by $0.9^{|H|-k}$.
- Spread activation ($\alpha=0.85$ and edge weights=1) such that only word vertices receive activation.
- Spread activation again ($\alpha=0.85$ and edge weights=1) such that only method vertices receive activation.
- Recommend methods with greatest activation.

Figure 3.2. Formal definition of the PFIS-R Algorithm.

each method being a document. The algorithm maintains a word-by-document matrix MW that specifies the importance of each word in W to each method in M by computing the TF-IDF (term frequency–inverse document frequency [Baeza-Yates & Ribeiro-Neto, 1999]) weight for every word-method combination. It uses MW to assess the lexical similarity between methods by constructing a document-by-document matrix MS that specifies the cosine similarity measure [Baeza-Yates & Ribeiro-Neto, 1999] for all pairs of documents. $MS(m)$ denotes the vector of cosine similarity scores associated with a particular method m in MS.

To account for the programmer's foraging momentum, TFIDF-R sums the $MS(m)$ vectors for the last δ methods to which the programmer navigated, decaying older navigations at a rate of 0.9. Specifically, for each method m such that m is the kth method in H, $decayedMS(m) = 0.9^{|H|-k} \cdot MS(m)$. A final recommendation vector VR sums the decayed

BUG: Problem with character-offset counter.

In the lower left corner of the jEdit window, there are two counters that describe the position of the text cursor. The first counter gives the number of the line that cursor is on. The second counter gives the character offset into the line.

The character-offset counter is broken. When the cursor is at the beginning of a line (i.e., before the first character in the line), jEdit shows the offset as 1. However, the offset should begin counting from 0. Thus, when the cursor is at the end of the line, it will display the number of characters in the line rather than the number of characters plus 1.

Figure 3.3. The text of the jEdit bug report.

Algorithm	Task period assignment for each participant								
	2	3	5	6	7	8	9	10	11
PFIS-R$(\delta=1)$	2nd			2nd			1st	1st	
PFIS-R$(\delta=10)$		2nd	2nd		1st	1st			2nd
TF-IDF-R$(\delta=1)$	1st		1st			2nd	2nd		1st
TF-IDF-R$(\delta=10)$		1st		1st	2nd			2nd	

Table 3.1. Treatment assignments to participants and task periods. (Participants 1 and 4 removed due to technical failures).

vectors for the last δ methods in H. The algorithm regards the methods with the greatest values in VR as the best recommendations, and those sort first in Figure 3.1.c.

3.3.2 Participants

We recruited 11 professional programmers from a large company to participate in an empirical study comparing the usefulness of our four different algorithm configurations. Technical failures in the study environment invalidated the data for two of these participants; thus, we analyzed only the data for the remaining nine participants. We asked each participant to fix a defect in the jEdit software project. jEdit is a text editor designed for programmers. None of the participants had ever seen the jEdit code before, and with 6,468 methods (98,652 non-comment lines of Java code), the jEdit code base provided a large space to forage within. The defect was from an actual bug report, #2548764, which described a problem with jEdit's text "folding" functionality. Figure 3.3 shows the details of the bug report.

3.3.3 Procedure

Each study session took about 180 minutes. Participants began by filling out a pre-questionnaire that gathered background information. Next, they engaged in two back-to-back task periods. Prior to the first task period, we introduced participants to the video equipment and started recording. Each task period was associated with a different treatment (i.e., one of our four algorithm configurations). Table 3.1 lists the treatment assignments, which we balanced to account for learning effects.

Each task period began with a short tutorial task on the tool. Some algorithms may have produced poor recommendations in the first task period, and we did not want participants who received such treatments to ignore recommendations in the second task period. Consequently, the tutorial in the second task period informed the participant that the tool's recommendation algorithm had been switched and asked the participant to repeat the tutorial task to see how the recommendations had changed.

Within each task period after the corresponding tutorial, participants worked on the jEdit debugging task for 35 minutes. To assess how participants used recommendations and when they were useful, we asked them to "talk aloud" as they worked. At the end of each task period, we interrupted participants and had them fill out a questionnaire that asked for their opinion of the recommendations.

After both task periods, the study session ended with a 35-minute semi-structured retrospective interview in which an interviewer stepped through events of interest in the screen capture videos and asked the participant questions about those events. For each time the participant navigated to a method, the interviewer asked, "What did you learn from this place?" and for each time the participant clicked on a recommendation, the interviewer asked the participant why he/she did that. Due to time constraints, the interviewer was sometimes unable to ask about all these events of interest in a session.

3.3.4 Analysis Procedure

Ultimately, a recommender's quality is its usefulness, but evaluating usefulness raises challenges. For instance, a participant may ignore recommendations that would otherwise be useful because the participant is unfamiliar with the tool. As professional programmers, our participants have honed their navigation strategies over years of experience, and they might favor their practiced strategies over a relatively unfamiliar tool.

To address this problem, we conducted two analyses. The first was a quantitative analysis of *hit rate*. This analysis compared the algorithms' recommendations to the places that participants actually navigated, regardless of whether they actually used the tool to navigate there. The second was a qualitative analysis of *demonstrated usefulness*, which assessed whether recommendations participants followed via the tool were useful in the participants' opinions.

Hit Rate

To assess hit rate, we computed the top-10 recommendations from the tool algorithm that was active at the moment before each user made each navigation. We scored the recommendations as a *hit* if the user navigated to a recommended location within the next s navigations. It was difficult to estimate the window of time in which a recommendation might have been usefully pertinent to a participant, so we explored two time-window sizes: a hit within the next 10 navigations ($s=10$) and a hit on the next navigation ($s=1$). Note that the time-window size s is concerned with how we score the hit rate of recommendations (over future navigations), not to be confused with the sensitivity to momentum factor δ in this study, which also had values of 1 and 10 (the number of past navigations from recent history).

Recall that nondeterminism arises in the case of ties between recommendations. To account for this nondeterminism, we took the best tie-breaking choice to make a best-case list of recommendations (maximizing hit rate), and we took the worst tie-breaking choice to make the worst-case list (minimizing hit rate), thereby bounding the effects of nondeterministic tie-breaking.

Given the two choices of s and the two choices of tie-breaking, we obtained four separate assessments of each recommendation algorithm's hit rate. Each of these was treated as a separate dependent variable in a separate regression. Because our dependent variable was dichotomous (indicating whether or not a recommendation was considered a hit), we used logistic regression. In addition to algorithm, our regression model also included task period and participant as categorical independent variables (i.e., indicator variables). After performing the regression, we used the Wald test to assess whether each variable (algorithm configuration, participant, and task period) had a statistically significant effect on hit rate. Using the coefficients provided by the regression, we also computed the overall hit rate of each algorithm, after subtracting out the effects of participant and task period.

Demonstrated Usefulness

To assess recommendation usefulness, we looked at the recommendations that the programmers followed to see if the recommendations helped the programmers make progress. To do this analysis, we qualitatively analyzed the verbalizations the participants made during task performance and their answers to our probes during the retrospective interview. In particular, for each followed recommendation, our interviewer asked the participant, “What did you learn from this place?” If the participant responded negatively (e.g., “I learned nothing”) or with apparent uncertainty (e.g., “I don’t know”), we coded the navigation as not useful; otherwise, we coded it as useful. Two researchers achieved an IRR of 100% over 20% of the data (Jaccard index) before splitting up and coding the participants’ responses individually.

3.4 Results

3.4.1 Hit Rate

Participants averaged one navigation every 90 seconds, or 447 navigations in all (Figure 3.4). We used this data to analyze hit rate. Figure 3.5 depicts the hit-rate results by algorithm configuration for window sizes $s=10$ and $s=1$. Our data revealed significant differences in the hit rates for the four configurations. Specifically, we consistently found $\chi^2(3) \geq 9$ and $p \leq 0.03$ regardless of whether we used $s=1$ or $s=10$, worst- or best-case tie-breaking, and $\delta=1$ or $\delta=10$.

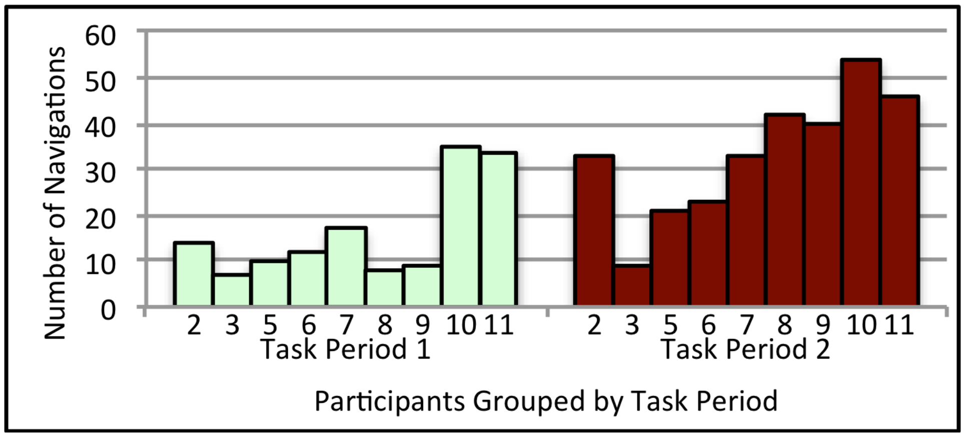


Figure 3.4. Number of navigations by each participant.

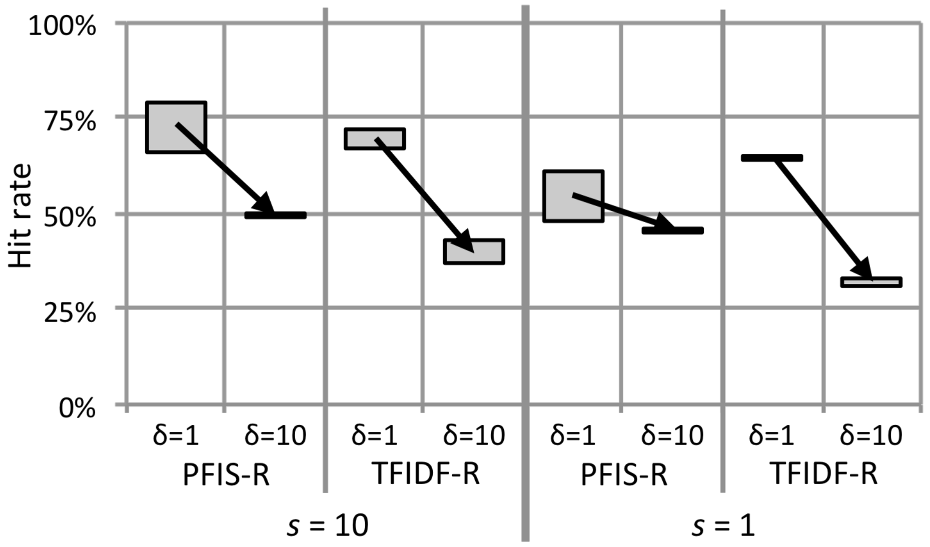


Figure 3.5. Hit rate of each algorithm (averaged over all task periods and participants) for *s*=10 (left half) and *s*=1 (right half), with rectangles indicating ranges between best- and worst-case tie-breaking. The *δ*=1 algorithms consistently out-performed the *δ*=10 ones.

Low vs. High Sensitivity to Momentum (RQ1)

Each low-momentum (*δ=1*) algorithm consistently scored a significantly better hit rate than the corresponding high-momentum (*δ=10*) algorithm (Figure 3.5). We found $z \geq 2.45$, $p \leq 0.01$ regardless of whether we used *s=1* or *s=10*, worst-or best-case tie-breaking, and TFIDF-R or PFIS-R. Thus, with respect to RQ1, we found that low-momentum algorithms outperformed high-momentum.

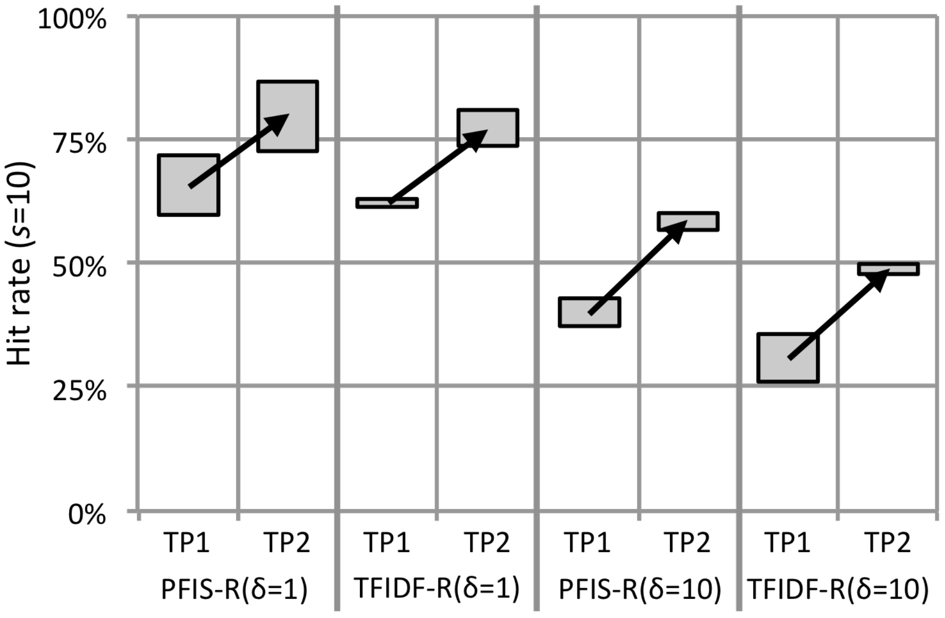


Figure 3.6. Hit rate (*s*=10) of each algorithm in the first (TP1) and second (TP2) task periods, with rectangles indicating ranges between best-and worst-case tie-breaking. Hit rates increased in the second task period (see arrows).

Word and Code-Structure Cues vs. Word Cues Alone (RQ2)

We found no meaningful difference between PFIS-R and TFIDF-R, regardless of whether we used $s=1$ or $s=10$, or worst- or best-case tie-breaking. Neither low-momentum ($\delta=1$) configuration of PFIS-R and TFIDF-R outperformed the other (at $p < 0.05$). Between high-momentum ($\delta=10$) configurations, PFIS-R outperformed TFIDF-R, but the difference was not statistically significant (at $p < 0.05$). Additionally, we found no significant interaction between algorithm and sensitivity to momentum (δ).

First vs. Second Task Period (RQ3)

Participants navigated almost twice as frequently during the second task period as during the first (Figure 3.4). Comparing between task periods (Figure 3.6), we saw suggestive improvements in accuracy when evaluating recommendations with respect to the programmer's next navigation ($s=1$), but these differences were not statistically significant (at $p < 0.05$). However, with respect to the programmer's next 10 navigations

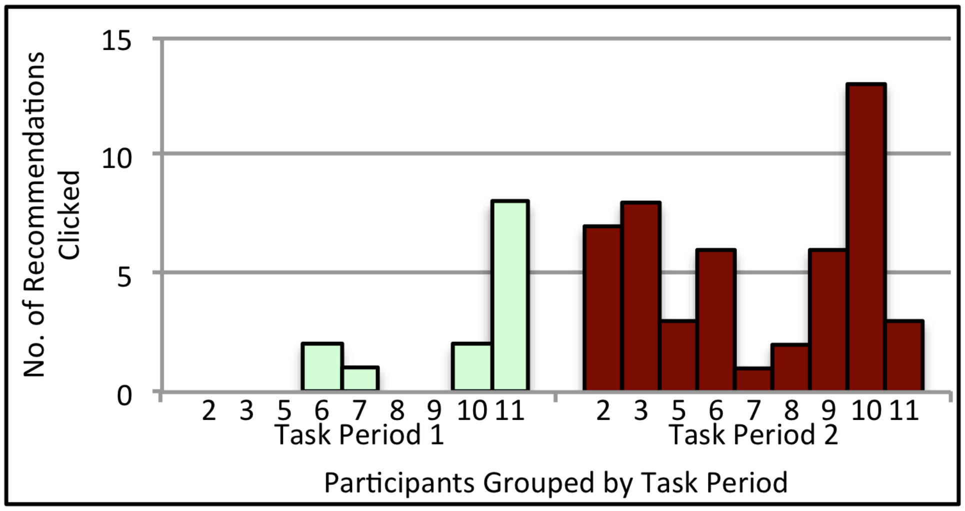


Figure 3.7. Number of times that participants clicked recommendations. Annotations indicate participant ID.

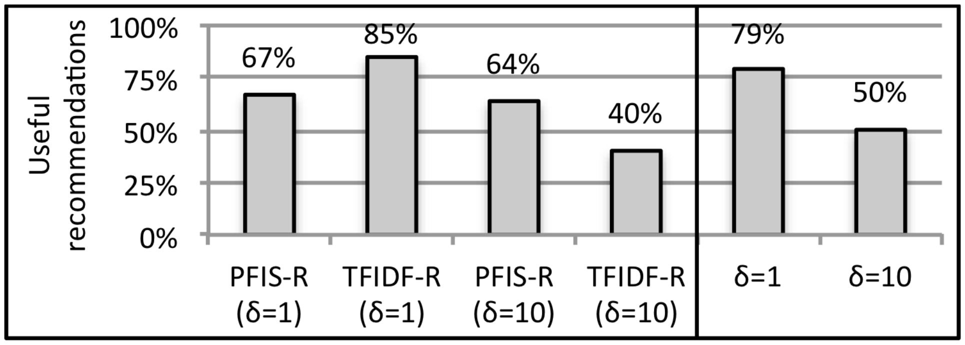


Figure 3.8. Percentage of recommendations that participants reported learning something from, grouped by algorithm and by low and high momentum.

(*s=10*), both algorithms improved significantly in the second task period over the first (pairwise $z = 2.37$, $p = 0.02$ for worst-case tie-breaking; $z = 3.44$, $p < 0.001$ for best-case).

3.4.2 Demonstrated Usefulness

All participants clicked some recommendations (Figure 3.7), for 62 clicks in total, and we collected interview responses for 44 clicks. Figure 3.8 summarizes the proportion of those clicks that participants reported as useful. We did not test for statistical significance due to the low number of clicks in each of the treatments.

Low vs. High Sensitivity to Momentum (RQ1)

The participants reported as useful a greater proportion of the recommendations from low-momentum algorithms than the recommendations from high-momentum algorithms (RQ1). This tendency triangulates with our finding that the low-momentum ($\delta=1$) algorithms demonstrated a higher hit rate than the high-momentum ($\delta=10$) ones.

Word and Code-Structure Cues vs. Word Cues Alone (RQ2)

Also triangulating with our hit-rate results, our demonstrated-usefulness results showed no consistent difference between PFIS-R and TFIDF-R.

First vs. Second Task Period (RQ3)

Similar to what we saw with hit rate, the participants clicked recommendations more frequently during the second task period than during the first.

3.4.3 Opinion Questionnaire

Figure 3.9 illustrates the opinion results. Participants completed a total of 18 opinion questionnaires (1 per task period) that asked the question (5-point Likert) "Was this tool valuable in getting you to useful parts of the source code?" Again, we omitted statistical tests when analyzing this data due to the low number of clicks in each treatment.

Low vs. High Sensitivity to Momentum (RQ1)

Triangulating with our hit-rate and demonstrated-usefulness results, participants rated the low-momentum algorithms more favorably than the high-momentum ones (Figure 3.9.b).

Word and Code-Structure Cues vs. Word Cues Alone (RQ2)

Similar to our hit-rate and demonstrated-usefulness results, the opinion results showed no suggestive difference between TFIDF-R and PFIS-R.

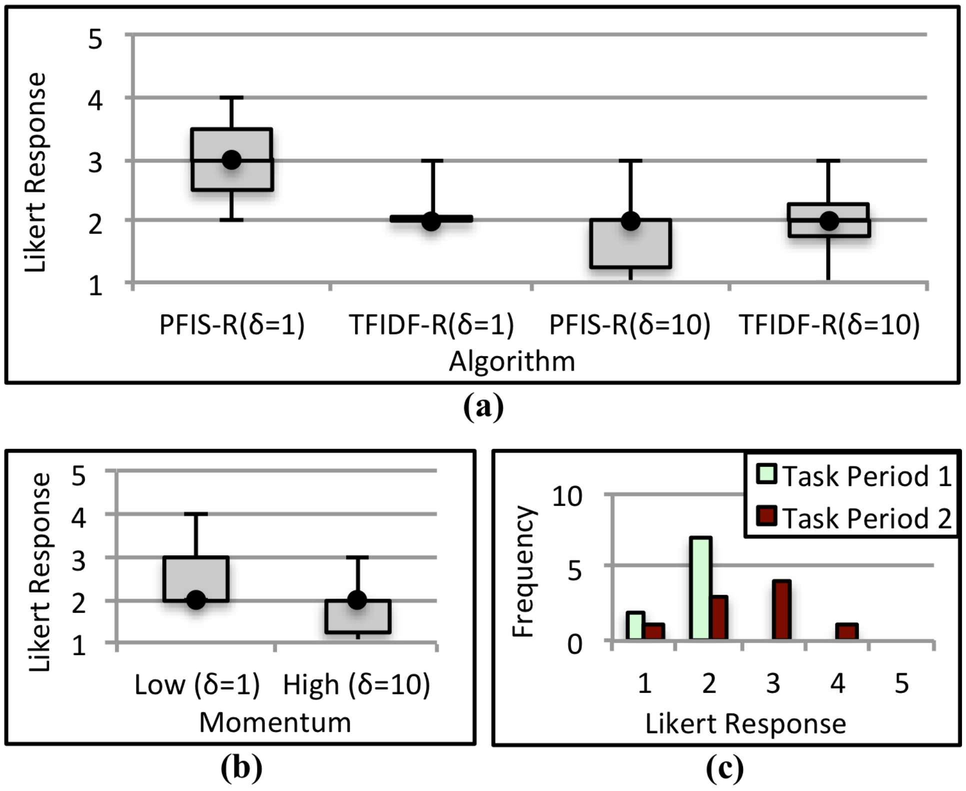


Figure 3.9. Results of questionnaires on the value of the recommendation system (1 = Entirely worthless, 5 = Very valuable).

<u>First vs. Second Task Period (RQ3)</u>

Participants seemed to rate all algorithms more favorably in the second task period than in the first (Figure 3.9.c), consistent with our hit rate analysis where the second task period had a higher hit rate than the first.

3.5 How Recommendations Were Used

To better understand when and why participants found recommendations useful (RQ4), we qualitatively analyzed the talk-aloud and retrospective interview recordings.

3.5.1 Using Recommendations for Efficiency

Benefits of Recommendations Based on Code Structure

Some participants benefitted from PFIS-R's use of code structure in making recommendations. For example, Participant 2 found code-structure-based recommendations useful while using an exception stack trace to navigate through code. He used the stack trace to open a method that contained a call to another method `recalculate-LastPhysicalLine` (i.e., following the call graph structure):

> Participant 2: "I wanted to go to the declaration of [`recalculateLastPhysical-Line`] and you know, my god, the [recommendation system] had it sitting there so I thought I'll go over and select it... I like that it was bigger and it was right there so it just seemed like I would both go to it and maybe learn something in the process..."

This quote reveals a case where a structure-based recommendation led directly to a desired method, making it efficient to navigate there in a single mouse click. PFIS-R's use of code-structure cues made a difference in this case. In contrast, running the TFIDF-R algorithm (with $\delta=1$ and $\delta=10$) on Participant 2's navigations revealed that this words-only algorithm would not have made this recommendation (even in the best case).

Recommendations as a Working Set

Several participants used recommendations to efficiently navigate back to previously visited methods; e.g.:

> Participant 9: "At this point, I'm kind of abusing the recommendations as a history because they are the fastest way to get where I want to go."

As another example, Participant 8 took advantage of the implicit "history" when, using Eclipse's debugger, he accidently stepped out of a method `transactionCom-plete` that he meant to inspect:

> Participant 8: "So that kicks me out to the catch and I remembered it was in the `trans-actionComplete` method... I had remembered that this recommendation had shown a bunch of `transactionComplete`'s, so I was just clicking around to just find where I was."

It could have taken Participant 8 considerably more effort to get back to `transactionComplete` (e.g., by rerunning the debugger) without the recommendation. Participants received historical recommendations from both low- and high-momentum algorithms because all algorithms made recommendations to previously visited methods.

Participants who used recommendations to get back to previously visited methods were essentially using the recommendations as a sort of working set. Previous research has shown that programmers tend to navigate frequently to methods in their working set [Parnin & Gorg, 2006; Piorkowski et al., 2011], and several successful tools have been developed that emphasize working set [Bragdon et al., 2010b; DeLine et al., 2005b; Kersten & Murphy, 2005]. A novel feature of our operationalization of Reactive IFT is that it implicitly supports working set while also helping the user explore new places.

3.5.2 Using Recommendations for Discovery

"Aha! Moments"

Some participants followed recommendations to methods that they were apparently unaware of and expressed excitement about how useful the recommendation turned out to be. For instance, Participant 2 was having difficulty finding code to focus on. He perused the recommendations:

> Participant 2: "'You might want to go here.' ... '`collapseFold`' ... '`expandFold`' ... OK, '`collapseFold`.' [Selects `collapseFold` recommendation; reads code comments.] 'Collapses the fold associated at the specified line index.' *OK! Now, this is where I want to be!* Collapsing the fold."

Participant 6 was having similar difficulty finding code to focus on when he turned to the recommendations:

> Participant 6: "[Selects `loadMenu` recommendation; reads code comments.] 'Creates a menu, the menu label is set from the name property, name.label propery.' *Oh! Oh! Yes!* ... This thing looks like the class that might put up the menu. *I like that!*"

Our choice to make half of the recommendations be to previously unvisited methods created opportunities for participants engaged in exploratory navigation to have "aha moments" such as these. However, not all exploratory navigations were to unvisited methods. Participant 6 had already visited `loadMenu` when he followed the recommendation. On his first visit, he did not notice anything interesting about the method. It was only after he followed a recommendation to the method that he discovered needed information in the method.

Misleading Cues

In some instances, participants navigated to methods that contained cues that generated strong scent, but the recommendations did not help them discover methods that would satisfy their goals. For instance, Participant 6 wanted to find the method that implemented the action for one of the items in jEdit's Edit menu. All the methods that he navigated to contained words (i.e., cues) related to his goal, such as menu, edit, and action. He then noticed a recommendation for `JEditAbstractEditAction` that included the keywords edit and action:

> Participant 6: "I was looking for the method that would get run when someone picked on that [Edit] menu item. So again, [the recommendation] could be the mnemonic suggestion of the class that maybe that was some kind of action that would get run in the Edit menu."

Participant 6 clicked the recommendation, but unfortunately the method did not contain code for implementing menu-item actions. Instead, it contained code that implemented the menu framework.

The recommendation system (running TFIDF-R, $\delta=10$) could not help Participant 6 because it could not relate the word cues that he was following to the method that would satisfy his goal. That method contained entirely different word cues from the menu-framework code. Because the participant was navigating through the framework code, building momentum on those cues, the algorithm did not recognize the relevance of the needed method.

Code structure cues used by PFIS-R might help overcome this problem generally; however, in this particular case, PFIS-R would not have had access to the structural information needed to connect the menu and action code because that information was contained in a properties file that was not part of jEdit's Java code base. This problem highlights the importance of having complete structural information in operationalizing Reactive IFT.

3.6 Discussion

3.6.1 Sensitivity to Momentum

The Reactive IFT tools that were less sensitive to participants' foraging momentum (i.e., low momentum, $\delta=1$) produced better recommendations than those more sensitive to momentum (i.e., high momentum, $\delta=10$). This result runs counter to previous work on recommender systems, which suggested that using more history was better for predicting future navigation [Lawrance et al., 2010; Piorkowski et al., 2011]. Outside software maintenance, successful recommender systems use historical behavior going back days, weeks, and years [Resnick & Varian, 1997]. In light of this past work, our result that $\delta=1$ produced better recommendations than $\delta=10$ was unexpected.

One interpretation is that this result may be due to participants' goals evolving rapidly and repeatedly. Studies have shown that as programmers navigate through code, they continually ask new questions [Ko et al., 2006; Sillito et al., 2005, 2006]. If participants' goals did change frequently, it would put the high-momentum operationalizations at a considerable disadvantage, because a high-momentum operationalization considers cues associated with a mix of goals, some of which are no longer relevant. We will return to this question in the next chapter.

These results underscore the importance of understanding a predator's foraging momentum in operationalizing Reactive IFT. For these programmers engaged in debugging, low-momentum was apparently better at inferring goals, but depending on the foraging context (e.g., government agents performing intelligence analyses or end users debugging spreadsheets), a higher foraging momentum may produce better outcomes. One limitation of the current study is that we examined only two values of δ. Future studies could further investigate the effect of momentum.

3.6.2 Changes in Foraging Behavior over Time

Changes in participants' foraging behavior as the task progressed may have been responsible for the algorithms' improved hit rate in the second task period. Studies have shown that programmers pursue different kinds of information [DeLine et al., 2005b; Sillito et al., 2006; Wiedenbeck & Evans, 1986] and engage in different types of activities [Lawrance et al., 2013] as a task progresses. In our study, the participants may have had difficulty finding strong scents in the earlier stages of debugging. As they foraged, they may have homed in on places with stronger scent. Since our algorithms relied on scentful cues to approximate goals, they might have been less accurate earlier in the task when the participants were navigating through low-scent methods, and became more accurate as scent increased.

One possible way to handle low-scent periods is to switch algorithms during such periods. For instance, an algorithm might detect changes in a user's foraging momentum and swap in the most appropriate algorithm for approximating the user's goal. Open questions for future research include how to detect a user's shifts in momentum and how to make recommendations in the absence of scentful cues.

3.6.3 Beyond Word Cues

Although not reaching statistical significance, the hit-rate results showed a tendency to favor PFIS-R over TFIDF-R, and this tendency triangulates with the suggestive difference between PFIS-R and TFIDF-R in the opinion questionnaire results (Figure 3.9.a). This tendency was consistent with the qualitative data. We observed many participants following call dependences while debugging. Moreover, we saw one instance (Participant 6) where a method that would have satisfied the participant's goal contained none of the words that the participant followed scent from.

The implication for tools is that when operationalizing Reactive IFT for a new context, the tool may need to consider other cues in addition to words. Word cues exclusively have dominated the work on web foraging, and they may be particularly effective in that context due to the large volume of unstructured natural language text that web pages contain. However, in a context like programming, where there is less natural language text and more structure, other types of cues may be valuable.

3.6.4 Predictive Factors as Scent

Given these results, we can now assess the assumption initially stated in this chapter, that the factors used by these tools are actually just different types of scent. First, consider the Lexical similarity factor, which has already been established by Pirolli as an approach for approximating programmers' scent.

Lexical Similarity has successfully predicted and explained how predators forage for information on the web [Chi et al., 2000, 2001; Fu & Pirolli, 2007; Pirolli, 2005; Pirolli et al., 2005; Pirolli & Fu, 2003] and in predictive models of programmer navigation [Lawrance et al., 2008b, 2013; Piorkowski et al., 2011]. Pirolli's rationale for lexical similarity as an approach for approximating scent rests on the facts that (1) humans use words extensively to communicate both explicitly and implicitly about information, and (2) that there is an abundance of lexical cues contained in both web pages and IDEs. As Pirolli put it:

> "Web page designs have evolved to associate (by human design or automated information systems) small snippets of text and graphics with such links. Those text and graphics cues are intended to represent tersely the content that will be encountered by choosing a particular link on one page and navigating to the linked page. When browsing the Web by following links, users must use these cues presented proximally on the Web pages they are currently viewing to make navigation decisions." [Pirolli, 2005]

Turning to Recency, our argument rests on the definition of scent, which is the predator's attempt to optimize actual value per actual cost, using their expectations of value and cost. We consider Recency in light of this definition of scent.

The high predictive accuracy of Recency (a.k.a. foraging momentum) suggests that programmers favor patches that they have visited before. A programmer revisiting previously visited patches can be explained as a way for the programmer to reduce both actual cost[1] and their expectations of that cost. Actual cost for *within*-patch foraging are likely to be reduced when the programmer has already processed at least part of the patch. Likewise, actual costs for *between*-patch foraging are likely to be reduced if the programmer remembers how to easily go back to that patch. In fact, many IDEs support programmers' revisiting of patches by providing affordances that enable programmer to quickly return to previously visited methods (e.g., the back button in Eclipse), a reduction of actual between-patch cost.

A key point linking these actual costs to scent is that the programmer has "inside information" about a patch's actual costs, if she has recently been to that patch before, and therefore has a reasonable idea of the cost of getting there and the cost of processing what is in there. Thus, Recency provides an advantage to programmers' ability to align expected cost with cost, thereby facilitating their efforts to optimize scent (expected value per expected cost) via the expected cost factor.

In certain situations, revisiting code may also help programmers increase the value of a patch. Consider one situation in which a programmer returns to previously visited patch: the "prey in pieces" problem (explained later in Chapter 6). In this situation, the programmer cannot find the answer to her foraging goal in a single location; she must collect the information from multiple patches and reassemble them. However, to understand the whole, she may need to revisit the relevant patches several times to build the context necessary to address the foraging goal that she had. With each revisit, she gains additional context, which may change and potentially increase the actual value[2] of the patch. Another example occurs when the programmer is stepping through a loop with a step debugger. Consider an off-by-one error. As the programmer steps through the loop

[1] Actual cost is a combination of the time it takes for a programmer to go to a new patch (C_b) and also the time it takes to process a patch (C_w). Actual cost depends on context, and can change if the context changes. For example, it depends on the predators' prey (by reducing the number of relevant patches to forage between or the types of information features to forage within), if predator previously enriched the environment to reduce cost, etc.

[2] Actual value is a measure of the information gained by the predator. Like actual cost, actual value can also change according to the programmer's prey (because the amount of information gained from an information feature changes based on the predator's foraging goal) or if the predator has previously enriched the environment.

into the iteration where the off by one occurs, the actual value of the patch may increase as the in-memory values of variables change and the defect becomes evident.

Much like cost, the programmer likely has an advantage in predicting the patch's actual value if she has recently seen the patch's contents before and remembers it. That knowledge facilitates the programmer's ability to align expected value with actual value.

We can also explain the predictiveness of code structure from a scent perspective in how programmers use IDE affordances that leverage structure to reduce actual cost. Structure is a fundamental aspect of program code, and understanding a program's structure is a goal that programmers are known to pursue [LaToza & Myers, 2010; Sillito et al., 2006]. Some questions related to understanding structure include locating an object's definition, understanding hierarchy or determining control flow. To efficiently answer these sorts of questions, programmers tend to follow affordances provided by most IDEs to follow structural links between patches that otherwise would not exist. By doing so, programmers reduce the actual cost of foraging because the alternative would be to navigate to all the relevant patches via other links that are less direct (thereby increasing actual cost). For example, Participant 2 leveraged the code structure factor in the recommender to efficiently navigate to a patch relevant to his current prey via a structural link.

Structural-link affordances provided by the IDE tend to be predictable, and programmers leverage that predictability to align their expectations of cost with actual cost. Consider the example of a programmer using an IDE affordance that opens an object's declaration. The programmer can easily align expected and actual between-patch cost using her prior experience. She expects to follow a structural link to get to the object's definition once she uses the affordance, and predictably, that is what happens.

By the same token, in the above example the cues on the structural links enable programmers to align their expected value with the actual value of the patch. This is because these affordances provide cues that align with the programmer's goal (in the above example, the object's definition) (e.g., "Open declaration").

By the above logic, we believe that these factors fulfill the definition of scent, and are different types of scent a programmer can follow: Lexical scent, Recency scent and Structure scent.

3.6.5 Scent as a Unifier for Software Engineering

The above demonstrates how viewing tools through this lens translates their approaches to supporting programmers' navigation to attempts to leverage these different types of scent. This in turn suggests that these tools could be rewritten (and new tools could be written) to use the same scent-based computational algorithm as a core, and thus unify the different tools under a single, IFT-based abstraction.

However, writing such an algorithm requires addressing several unanswered questions regarding the different types of scent. For example, how does a programmer decide which type of scent to invest in at a given time? One likely possibility is that this decision is influenced by the prey that the programmer is foraging for at that moment. But that begs the question of what is the relationship, if any, between the programmer's prey and the type of scent they follow? Furthermore, although we have provided some evidence of how the factors may affect the programmer's expectation of value and cost, we believe there is still a knowledge gap not only in terms of how a programmer estimates of value and cost change at any given moment, but also what other types of scent may be influencing those estimates.

Successful tools also shed light on other types of scent that are relevant to programmers. In this chapter, we identified and discussed three types of scent, but as new tools are developed and evaluated, we expect additional types of scent to emerge.

3.7 Summary of Results

This chapter investigated how to bring Reactive Information Foraging Theory to recommender tools for information-intensive, ill-structured problems. The algorithms we investigated with professional programmers in a debugging task showed that:

- RQ1: Surprisingly, assuming low foraging momentum (using only one navigation to inform its choices) produced better recommendations than those produced by assuming high momentum (the past ten navigations).
- RQ2: There was suggestive but inconclusive evidence that participants found the tool to be more valuable when it used words + structure than when it used words only.
- RQ3: The tool's recommendations were more useful to participants later in the task, suggesting that tools may need to be sensitive to shifts in users' foraging behavior.
- RQ4: Recommendations helped participants by revealing useful places that the participants were unaware of and also by facilitating navigation to known places.

Most important, this chapter has demonstrated how IFT's scent construct can unify SE research on tools to support programmer navigation. We explained how the factors underlying the various tools actually supported different types of scent that programmer followed. We also evaluated several factors in situ via a recommendation tool to shed light on what types of factors may be most relevant for a unifying scent-based algorithm. In the next chapter, we continue the theme of how IFT's constructs can unify SE by investigating types of prey and the strategies that predators use to pursue them.

Chapter 4 Foraging Diet: Unifying Prey and Foraging Strategies

Similarly to how we demonstrated how scent can unify several papers on tools to support programmer navigation under the IFT construct of scent, in this chapter, we consider using the IFT construct of prey as a unifier of research works on programmers' strategies during debugging.

To do so, we investigate how programmers' foraging goals affected their foraging behavior. Specifically, we ran a study that identified the types of foraging goals that programmers pursued and what strategies programmers chose to pursue those goals. This notion is captured in IFT as *foraging diet*, which is simply the prey that a predator chooses to pursue.

Diet so far has been mostly untapped in IFT. A notable exception is Evans and Card [Evans & Card, 2008b], who investigated the diets of web users who were "early adopters." They discovered that these users' diets were considerably different from the information commonly provided by mainstream news sites, and they identified the niche topics that made up the users' diets. They also noted that the information sources chosen by these users reduced the cost of attention by lowering the cost of social foraging and social interpretation. Clearly, these findings have strong implications for the design of sites to support such users. The Evans and Card work demonstrates the potential benefits of applying information foraging ideas to understand the diets of people in particular contexts.

Inspired in part by the Evans/Card paper, this chapter aims to expand our understanding of IFT diet by investigating the diets of professional programmers engaged in debugging. Work in the software engineering (SE) literature has investigated related ideas, such as the questions that programmers ask (e.g., [Fritz & Murphy, 2010; Ko et al., 2006; LaToza & Myers, 2011; Sillito et al., 2006]), but that work was not grounded in a theory, such as IFT. Thus, by investigating the information diets of professional programmers from an IFT perspective, our work aims to help bridge the gap between such results from the SE literature and the IFT foundations and results from the HCI literature.

For an understanding of the "what's" of diet to be truly useful, we also need to understand the "how's". Toward this end, we also investigate, from an IFT perspective, the *strategies* that programmers use during foraging. The literature contains numerous works on program debugging strategies (see [Romero et al., 2007] for a summary), but these have not been tied to IFT. We believe that such strategies both influence and are influenced by programmers' diets, and this chapter investigates these ties. These ties will serve as the foundation through which we leverage IFT to unify work on programmer diets and their foraging strategies.

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

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

4.1 Background

In the domain of software development (and especially debugging), information foraging often occurs in the context of sensemaking. The *sensemaking process* in an information-rich domain has been represented as a series of two main learning loops: foraging for information, and making sense of the foraged information [Pirolli & Card, 2005]. In this model, the role of IFT is central. In fact, in Grigoreanu et al.'s sensemaking study of end-user debugging [Grigoreanu et al., 2010] (which applied the Pirolli/Card sensemaking model [Pirolli & Card, 2005]) found that the foraging loop dominated the participants' sensemaking process.

In the software engineering community, there has been recent research focused on supporting the questions programmers ask [Fritz & Murphy, 2010; Ko et al., 2006;

LaToza & Myers, 2011; Sillito et al., 2006], and these questions can be viewed as surrogates for programmers' information goals. The software engineering analyses and tools have not been grounded in theory, but their empirical success shows that they are useful. A premise of this chapter is that IFT may be able to provide a richer, more cohesive understanding of programmers' information seeking behaviors than atheoretic efforts. Recently, we and a few others have begun investigating the efficacy of using IFT to understand programmer information-seeking (e.g., [Lawrance et al., 2008b, 2010, 2013; Niu et al., 2011; Piorkowski et al., 2012]). However, that work focused only on how programmers respond to cues. This chapter instead investigates the what's and how's of their diets, i.e., the relationship between programmers' information goals and debugging strategies.

A predator's *information goal*[3] defines her "ideal" diet, but what predators actually consume depends also on what information is available in the environment and how costly the information is to obtain. The relationship between cost and diet in IFT is explained well by Anderson's notion of *rational analysis*, which is based on the idea that humans tend toward strategies that optimally adapt to the environment [Anderson, 1990]. For example, it may make more sense for a programmer to forage for lower value prey if they find it too expensive to collect the higher value prey (even if that's in the "ideal" diet).

To help satisfy their diets, predators commonly engage in *enrichment*, that is, transforming the environment to facilitate foraging. For example, by searching on the Web, the predator enriches the environment by creating a new patch of search results, which could potentially satisfy some or all of the predator's information goals. In addition to using search tools, other examples of enrichment include writing a to-do list on a piece of paper and running a test on a program to create a patch of relevant program output. This work is the first to look at the role of enrichment (and other foraging strategies) for satisfying programmers' foraging diets in the domain of software debugging.

[3] Recall that a goal is set of information features that match prey available in the environment.

4.2 Methodology

4.2.1 Study Data

To investigate our research questions, we analyzed a set of nine videos we collected in a previous study of professional programmers debugging in an Eclipse environment [Piorkowski et al., 2012] (Chapter 3). In that study, the programmers used the usual Eclipse tools, plus a new IFT-based code recommender tool powered with a variety of recommendation algorithms. This set-up is consistent with real-world scenarios in which programmers work on unfamiliar code, such as a new team member being brought "onboard" a project, a programmer on a team needing to work on code that another team member wrote, or a newcomer to an open-source project.

To summarize the study setup, each video included screen-capture video, audio of what the participant said, and video of the participant's face. Participants "talked aloud" as they worked. Their task was to fix a real bug in the jEdit text editor[4]. None of the participants had seen the jEdit code before, and with 6,468 methods, it provided a large information space in which to forage. Each debugging session lasted two hours with a short break halfway through. No participants completed the task, and all exhibited instances of foraging throughout the two hours.

Categorization Procedures

We used a qualitative, multi-part coding approach to analyze these videos. First, we segmented the videos into 30-second intervals, resulting in roughly 70 segments per video. (We chose 30 seconds to be long enough for participants to verbalize a goal.) We then coded each segment to identify (1) instances of foraging, (2) participants' information goals, and (3) participant debugging strategies, allowing multiple codes per segment. To enhance generalizability, these code sets were drawn from prior studies, as we describe below.

To ensure reliability, we followed standard inter-rater reliability practices. Two researchers first worked together on a small portion of the data to agree on coding rules.

[4] See Chapter 3 for the bug's details.

They then independently coded 20% of the segments to test the agreement level. We computed agreement using the Jaccard index, as it is suitable when multiple codes are allowed per segment, as in our case. We performed a separate coding pass (with separate reliability checks) for each code set. For each pass, agreement exceeded 80%, so the two researchers then divided up the coding of the remaining data.

Information Foraging Behavior Codes

To code whether a participant showed evidence of information foraging within a 30-second segment, we used a two-part coding process. First, we segmented around participants' utterances and coded the segments. The codes were *foraging-start*, *foraging-end*, and *foraging-ongoing*. This code set was inspired by the scent-following code set used in [Lawrance et al., 2013], but ours focused only on whether or not foraging occurred, and not whether scent was lost, gained, etc. We coded an utterance as *foraging-start* when participants stated an intention to pursue a particular information goal and then took accompanying action to seek that goal, like searching. We coded an utterance as *foraging-end* when participants stated that they had learned some information, or expressed giving up on a goal. We coded an utterance as *foraging-ongoing* when participants restated a previously stated goal, or said they were still looking for something.

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

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

Table 4.1. Information goal type examples.

Information Goal Codes

We based the Information Goal code set on Sillito et al.'s empirically based taxonomy of 44 questions programmers ask, which Sillito et al. had grouped into four types [Sillito et al., 2006]. We coded the 30-second segments against the 44 questions, and then grouped them into the four types for presentation brevity. Table 4.1 lists the types, with a few examples of the Sillito questions that were our actual code set. We chose the Sillito questions for several reasons. First, they are a good fit for the program-debugging domain, because they categorize information needs specific to programmers. Second, they seem generalizable to a broad range of programming languages and environments, since Sillito et al. collected them from a study that covered seven different programming languages and at least eight different programming environments. Third, they are consistent with information goals identified in other studies from both programming and non-programming domains (e.g., [Grigoreanu et al., 2010; Hearst, 2011; Lawrance et al., 2013; Pirolli & Fu, 2003]). Finally, they are specific and low-level, enabling a code set with the potential for high inter-rater reliability.

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

Pattern	Example	Formal Definition
Oscillate: Back and forth between two adjacent types repeatedly.	1121212212	*O*= O_1 | O_2 where: O_1=*UpDn*(1,2) | *UpDn*(2,1) O_2=*UpDn*(2,3) | *UpDn*(3,2) *UpDn*(*a*,*b*)=*a*+b+(*a*+*b*+)+*a**
Stairstep: From 1 up through adjacent types to at least 3.	1122223	*Stairstep*= (1+2*)+ (2+3*)+ 3
Restart: Jump off the Stairstep down to 1	112331	*Restart*=*Stairstep* 1
Pyramid: Up then down the stairsteps. Constraint: If Pyramid, then not Stairstep.	12321	*Pyramid*= *Pup Pdown* | 2+ *Pup Pdown* 1 where: *Pup*=(1+2*)+(2+3*)+ *Pdown*=(3+2*)+(2+1*)+
Repeat: One type at least 10 times.	1111111111	*Repeat* = 11111111111* | 22222222222* | 33333333333* | 44444444444*

Table 4.2. Information goal patterns. Each definition is a regular expression of Goal Type instances. (+ means 1 or more instances, * means 0 or more instances, | means or). E.g.: 1+2+ means one or more instances of Type 1, then one or more instances of Type2. We omit Type4s next to Type 3s because 4 never followed 3 in our data.

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

Information Goal Patterns

To investigate how information goals relate to each other, we categorized the information goal data into the five patterns in Table 4.2. Four of the patterns (*Stairstep*, *Restart*, *Pyramid*, and *Oscillate*) came from literature suggesting progressions in these sequences (e.g., [Grigoreanu et al., 2012; Pirolli & Card, 2005; Sillito et al., 2006]). The fifth pattern, *Repeat*, emerged as a common pattern during the course of our analysis.

Following the Table 4.2 definitions, we used a greedy pattern-matching algorithm (which always returned the longest possible matches) to identify instances of the patterns in the goal data. We did not allow matches that contained a gap of 5 or more minutes (i.e., 10 or more 30-second segments) between goal utterances or contained an interruption/intervention, such as the between-session break. We permitted overlapping patterns, except for instances of *Oscillate* completely contained within a *Stairstep* or *Pyramid,* and for instances of *Stairstep* completely contained within a *Pyramid*. We omitted *Oscillate* and *Stairstep* instances in these cases, because they were essential components of the containing patterns. A single author performed this analysis because the definitions were objective and the analysis automatable.

Debugging Strategy Codes

To code participant strategies, we reused Grigoreanu et al.'s debugging strategy code set [Grigoreanu et al., 2010]. We chose these strategy codes because, while being specific to the program debugging domain, each also maps cleanly to one of the three key foraging activities [Pirolli & Card, 1999] within-patch foraging, between-patch foraging, and enrichment. (Technically, enrichment is a subset of within- and between-patch foraging activities; however, in this chapter, we use the term *within-patch foraging* and *between-patch foraging* to include only non-enrichment activities.)

Table 4.3 lists the strategy codes grouped by type of foraging activity. The Within-Patch strategies all involve looking for information within the contents of a single patch, such as in a Java method or web page. The Between-Patch strategies all involve navigating between different patches by selecting and clicking links, such as those provided by the recommender tool. The Enrichment strategies all involve manipulating the environment to facilitate foraging, for example, by creating a new patch of search results.

For each segment, we looked for evidence of the participant applying each strategy using indicators such as those shown in Table 4.3. A segment could have multiple strategy codes. Using this scheme, two coders achieved 80% agreement on 28% of the data, and then divided up the remaining data.

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

Table 4.3. Debugging strategy code set with example indicators for each strategy.

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

Table 4.4. Participants spent a large fraction of their time, ranging from 38% to 71%, foraging for information.

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

Table 4.5. Number of segments spent on the (codeable) types of information goals. Gray highlights each participant's must-pursued goal type.

4.3 Results

4.3.1 Preliminaries: How Much Foraging Did Participants Do?

As Table 4.4 shows, participants spent 50% of their 2-hour sessions foraging on average. We were unable to find prior measures of programmer foraging with which to compare this result, but Ko et al. measured time spent on *mechanics* of navigation. Their programmers spent 35% of the time on "the mechanics of navigation between code fragments" [Ko et al., 2006]. Even our participant who foraged the least still did so more than 35% of the time.

4.3.2 RQ1: The What's of Programmers' Diets

A Diversity of Dietary What's

Although all participants had the same high-level information goal (to find the information needed to fix the bug), their dietary preferences were diverse, as Table 4.5 shows. (Recall the four goal types defined in Table 4.1.) In aggregate, participants pursued the most goals of Type 1-initial, with slightly fewer in 2-build, and many fewer in the more complex 3-group and 4-groups. However, most participants did not conform to the aggregate: Only P6 and P9 had goal counts consistent with the aggregate. Instead, participants diets varied considerably.

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

Table 4.6. Frequency of pattern instances exhibited by each participant. The numbers in parentheses indicate the type of goals within the pattern (e.g., 1(3, 2) in the Oscillate row indicates patterns like 33322322, as defined in Table 4.2).

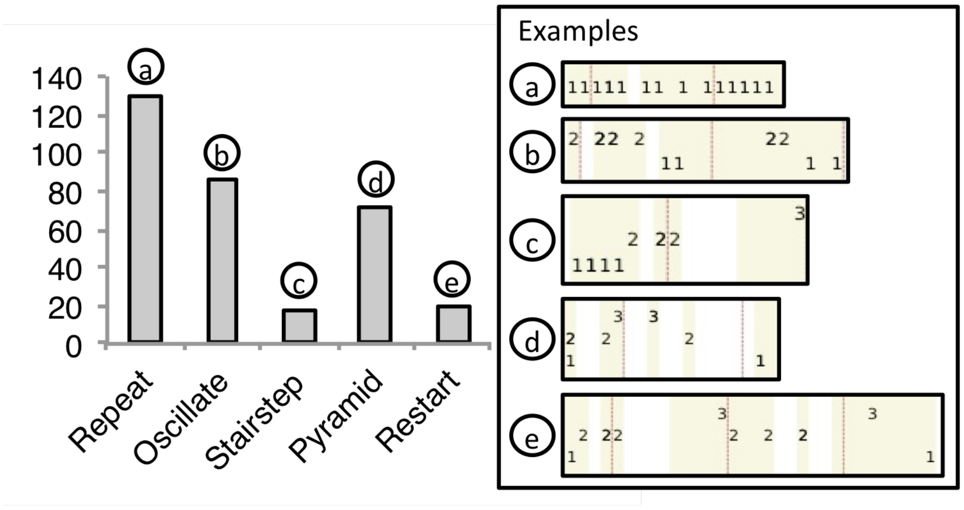


Figure 4.1. Frequency of goal patterns. Y-axis is the count of segments in each pattern. Each bar is labeled with an example from the participants' videos. The beige background denotes foraging; white is non-foraging (e.g., studying the code that has been found); and number denote the goal types.

Patterns of Dietary Relationships

Despite their dietary diversity, the progression of information goals that participants pursued often followed certain patterns (summarized in Table 4.6 and Figure 4.1; patterns defined in Table 4.2). Eight of the nine participants displayed one or more of the patterns, and 58% of segments in which a participant expressed a goal were part of a larger pattern. Participants exhibited a median of 1.5 patterns each, with P6 exhibiting all five.

For example, P6's use of the Restart pattern at the end of a Stairstep is shown in the Figure 4.1.e example. The Restart occurred when his Stairstep progression culminated in gaining the information he sought about the `handleMessage` method's relationship to the editor (a Type 3-group goal):

P6: "So this (`handleMessage`) is handling some events for the editor."

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

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

Some of these patterns were predicted by the literature. Sillito et al. [Sillito et al., 2006] suggested one progression: find an initial focus (1-initial), then build on it (2-build), then understand a group of related foci (3-group), and finally understand groups of groups (4-groups). Other empirical studies have found a similar progression from 1-initial to 2-build, including previous work on information foraging during debugging (characterized there as "debugging modes") [Lawrance et al., 2008b], and earlier work on how people seek information in web environments (summarized in [Hearst, 2011]). Furthermore, the notion of progressing from Type 1-initial to 2-build to 3-group to 4-groups is consistent with prior results from applying Pirolli and Card's sensemaking model [Pirolli & Card, 2005] to intelligence analysts and to end-user debuggers [Grigoreanu et al., 2012].

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

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

Why did participants exhibit the above patterns? To answer this question, we need two pieces of information: what strategies they used for their foraging, and how those strate-

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

Table 4.7. Usage (segment counts) of each strategy during foraging. Gray cells indicate the maximum frequency by participant and by strategy category. Although participants foraged in a total of 660 segments, the overall total of strategy segments (992) is greater because participants used multiple strategies during some segments.

gies came together with their goals and goal patterns. We discuss each of these in turn in the next two sections.

4.3.3 RQ2: The How's: Strategies during Foraging

Recall from Methodology (Table 4.3) that each debugging strategy maps to an IFT activity: within-patch foraging, between-patch foraging, and enrichment. Table 4.7 shows each participant's strategy usage by IFT category.

<u>Debugging Strategies Meet IFT</u>

Since much of the prior IFT research has focused on between-patch scent following (e.g., [Chi et al., 2001; Lawrance et al., 2010]), we were surprised that only 24% of participants' foraging fell into that category. Participants spent considerably more time foraging within patches and performing enrichment.

As Table 4.7 shows, participants used a diverse mix of strategies (median of 8 different strategies); however, each foraging category had clearly dominant strategies. Spatial was the participants' primary Within-Patch strategy; Control Flow was their primary Between-Patch strategy; and Code Search and Testing were together (but especially Testing) their primary Enrichment strategies.

What Participants Used Enrichment For

Enrichment is an activity wherein the predator changes the environment to facilitate foraging [Pirolli & Card, 1999]. The participants changed their environments in two ways. Code Search, Seek Help-Search, and To-Do Listing involved creating a patch of *links* to other patches for the predator to *navigate*. In contrast, Testing involved creating patches of information *content* for the predator to *process*.

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

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

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

> P5: [Looks at the debugger's Variable Watch view.] "*lineCount* is zero." [Reads code.] "I'm going to step into that (method)" [Steps.] "*count* is greater than–now *count* is zero. [Steps again.] "I'm stepping through the code. ... I'm trying to understand what this code is doing."

Despite prior findings about users' preference for searching (e.g., [Brandt et al., 2009]), four of the nine participants used neither Code Search nor Seek Help-Search. This lack of searching cannot be because the task was too easy (no one finished) or the

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

Table 4.8. These 6 strategies (out of 12) stood out. Each of these was used by everyone, was at least one person's most-used strategy, or was the top strategy for an IFT category.

code base was too small (it had 6468 methods). However, earlier findings on web information processing [Hearst, 2011] may explain this result. Hearst points out that, in many cases, browsing works better than searching because it is mentally less costly to *recognize* a piece of information than it is to *recall* it, and recall is often needed to formulate an effective search query. Consistent with Hearst's observation, every participant used the Code Inspection strategy.

Go-To Strategies for Foraging

Reconsidering Table 4.7 from a most-used perspective, some strategies stand out as having been used particularly often for one or more aspects of foraging. The leftmost four (white) columns of Table 4.8 summarize.

4.3.4 RQ3: What's Meet How's: Dietary Strategies

Strategies by Goal Type

Table 4.9 and Figure 4.2 tie all 12 of the strategies back to the participants' dietary goals. As the table and figure show, some strategies were strongly tied to particular goal types. For example, Specification Checking was used only for Type 1-initial goals, and Code Inspection was used primarily for Type 2-build goals. Figure 4.2 shows that

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

Table 4.9. Strategy usage by goal types. Gray highlights the maximum strategy usage for each goal type. The overall total (719) is greater than the total foraging segmetns (660) because some segmetns contained multiple strategies. The total for Seeking Help-Search was 0 because none of the strategy's Type 4 goal instances co-occurred with a goal statement.

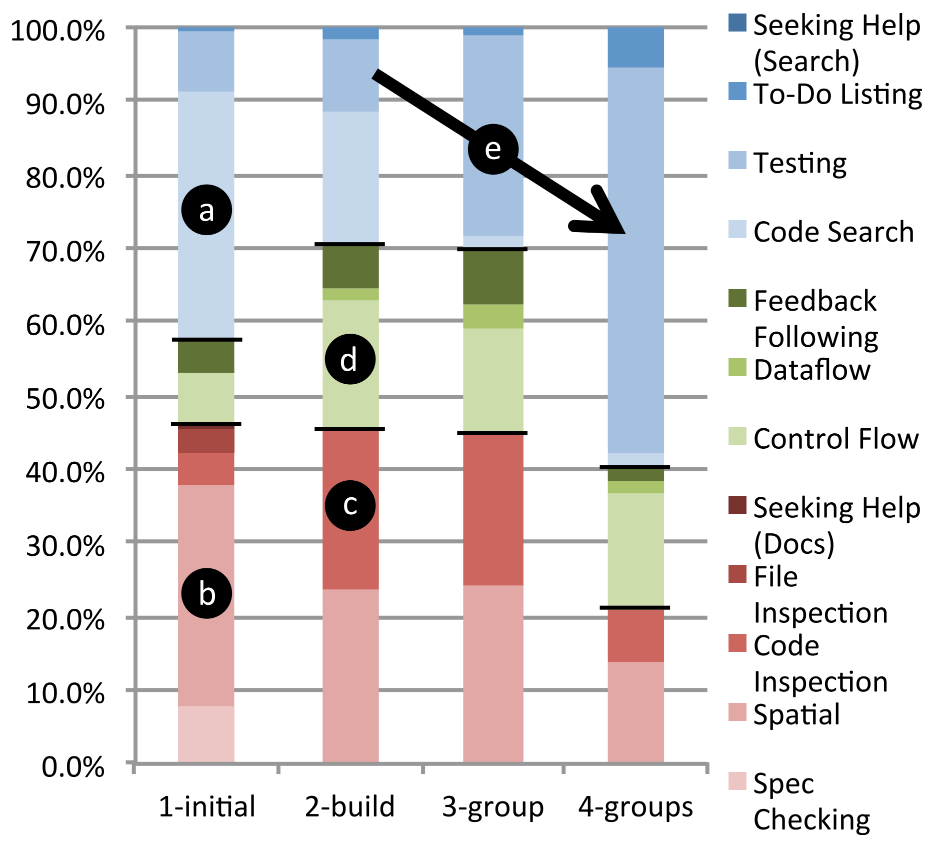


Figure 4.2. Strategy proportions by goal type. Strategies are color-coded, with black bars separating the IFT categories. Red: Within-patch. Green: Between-Patch. Blue: Enrichment.

participants used Code Search (labeled a) and Spatial (labeled b) more than the other strategies with their Type 1-initial goals. From a patch perspective, Spatial seemed particularly suited to helping participants cope with large patches, and Code Search with large spaces of patches. For example, P6 spent considerable time performing Spatial in the Package Explorer view (a patch containing hundreds of lines), looking for a Java class on which to focus:

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

P3, on the other hand, applied Code Search to search the 6468 methods for code related to deleting lines in jEdit:

> P3: "I would imagine that I would look for the word 'delete' perhaps, especially given that that's the term that's used in the menu." [Executes a search for *delete*.]

Participants tended toward different strategies for the Type 2–4 goals, which express progressively deeper relationships among code entities. For example, Figure 4.2 shows the shift away from Code Search and Spatial, and toward Code Inspection (c) and Control Flow (d) for Type 2-build and Type 3-group goals. Testing in particular (e) increased markedly from Type 2-build to Type 4-groups goals.

Considering participants' goal patterns in the context of their strategies (summarized in Table 4.10) sheds additional light on why the patterns emerged.

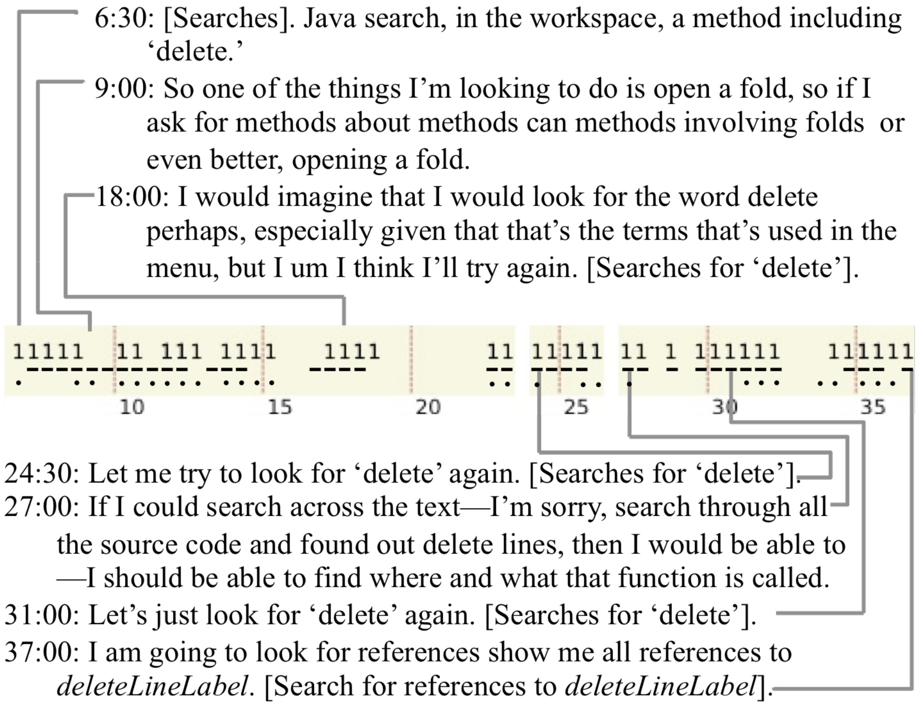
Pattern Repeat: Constant Goal Type, Constant Strategies

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

As an example of the first case, P7 followed the Repeat pattern on Type 2-build goals using three strategies continuously: Spatial, Code Inspection, and Control Flow. Eclipse

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

Table 4.10. Percentage of goal-pattern segments that co-occurred with each category of strategy. Recall that multiple strategies were allowed per segment. Gray denotes the maximum category for each pattern.



6:30: [Searches]. Java search, in the workspace, a method including 'delete.'

9:00: So one of the things I'm looking to do is open a fold, so if I ask for methods about methods can methods involving folds or even better, opening a fold.

18:00: I would imagine that I would look for the word delete perhaps, especially given that that's the terms that's used in the menu, but I um I think I'll try again. [Searches for 'delete'].

24:30: Let me try to look for 'delete' again. [Searches for 'delete'].

27:00: If I could search across the text—I'm sorry, search through all the source code and found out delete lines, then I would be able to —I should be able to find where and what that function is called.

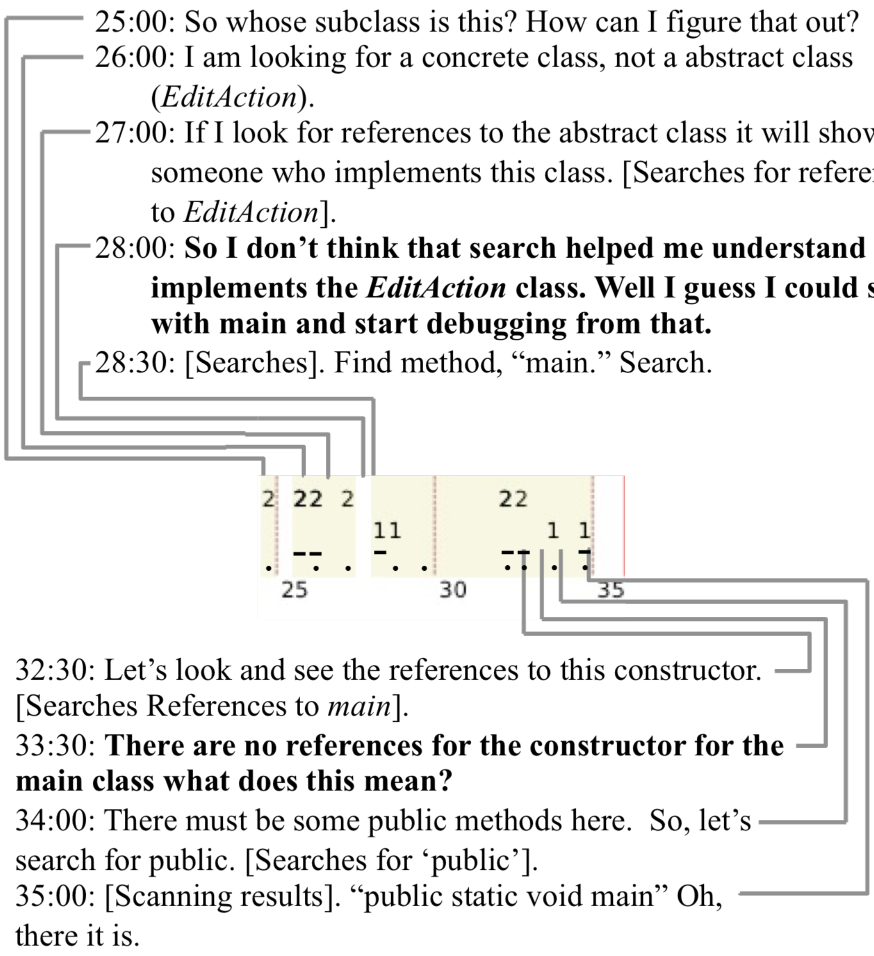
31:00: Let's just look for 'delete' again. [Searches for 'delete'].

37:00: I am going to look for references show me all references to *deleteLineLabel*. [Search for references to *deleteLineLabel*].

Figure 4.3. P3 continuously used Code Search (underlined) to find code relevant to deleting lines of text. He often complemented it with Spatial (dots). The beige background denotes foraging: white is non-foraging and the numbers indicate the minutes in the session.

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

When participants followed the Repeat pattern on goals of Type 1-initial or of Type 4-groups, their strategies were still constant, but not as fruitful. In the cases involving Type 1-initial, participants used Code Search (Enrichment) and Spatial (Within-Patch) extensively, but not particularly fruitfully, looking for a place to start. For example, Figure 4.3 shows P3 repeatedly using Code Search to find an initial starting point. Likewise, in P11's use of Repeat on his Type 4-groups goals, he used Testing across numerous segments of the pattern, trying to understand the relationship between changes he had made and the rest of jEdit's functionality. He pieced the information together by laboriously gathering it in small bits, one execution of the program at a time.



25:00: So whose subclass is this? How can I figure that out?

26:00: I am looking for a concrete class, not a abstract class (*EditAction*).

27:00: If I look for references to the abstract class it will show me someone who implements this class. [Searches for references to *EditAction*].

28:00: **So I don’t think that search helped me understand who implements the *EditAction* class. Well I guess I could start with main and start debugging from that.**

28:30: [Searches]. Find method, “main.” Search.

32:30: Let’s look and see the references to this constructor. [Searches References to *main*].

33:30: **There are no references for the constructor for the main class what does this mean?**

34:00: There must be some public methods here. So, let’s search for public. [Searches for ‘public’].

35:00: [Scanning results]. “public static void main” Oh, there it is.

Figure 4.4. The Oscillate pattern for P6. The abandonment of goals is highlighted in bold. The underlines are segments with Code Search. The dots are segments with Spatial. Note that strategies alternate with the goal types.

Pattern Oscillate: Changing Strategies to Dig Deeper

For the participants who followed the Oscillate pattern on Type 1-initial and 2-build goals, the story was similar to Repeat on Type 1-initial, except the oscillators tended to seek additional information from their search results. In particular, the oscillating participants would typically do a code search, explore the results a bit, decide they were on the wrong track, and return to searching. Unlike the Repeat pattern, the participants we observed within the Oscillate pattern switched strategies rapidly along with their goals. Figure 4.4 illustrates this behavior for P6.

Patterns for Enrichment and Goal Switching

Table 4.10 suggests that Enrichment tended to drive the interrelated Pyramid (up then down the stairs), Restart (stairs followed by starting again), and Stairstep (climb the stairs) patterns. Participants following the Pyramid pattern used the Enrichment strategies of Code Search and Testing equally often, but P6's instances of Stairstep and Repeat were characterized by almost exclusive use of the Code Search strategy. (Only P6 followed these two patterns.)

All three patterns were characterized by rapid goal fulfillment followed by a rapid switch to the next goal. This rapid fulfillment and initiation of the next goal type is consistent with our previous findings pointing to the reactiveness of foraging in this domain [Lawrance et al., 2010; Piorkowski et al., 2012] (Chapter 3).

The Most-Used Strategies' Strengths

This brings us to the particular strengths of different strategies. Refer back to Table 4.8; the rightmost (shaded) columns include the goal types and patterns we have just discussed for the most-used foraging strategies. As the table shows, certain classic debugging strategies were used heavily in *foraging* but often were concentrated into dietary niches. For example, Code Inspection and Feedback Following were generalists—used by everyone, but not the top in any particular IFT category, any goal type, or any pattern. In contrast, Code Search was a specialist, dominating some of the patterns and one of the goal types, but still used by only half the participants.

4.4 Discussion

4.4.1 The Long Tail of Diet What's

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

Our participants' diverse diets are reminiscent of the highly varied and personal diets reported by the Evans/Card study [Evans & Card, 2008b]. Evans and Card attributed this finding to a "long tail" demand curve, in which an abundance of available information makes it possible for people to satisfy their own individual, even quirky, desires for information. However, in the Evans/Card study, people foraged as part of their own individual tasks. Interestingly, we saw the same phenomenon with our participants, even though they all had the *same* overall goal (to fix the bug).

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

4.4.2 Scent's Role in Foraging Diets and Strategies

Recall that both goal types and strategies were derived directly from IFT's notions of prey and types of foraging. Goal types stood in for prey, the set of information features that a predator is looking for. Goal types provided a way to categorize questions that participants' asked during debugging. Similarly, strategies were a way to map actions that participants took in the IDE to one of the three types of foraging specified by IFT: within-patch foraging, between-patch foraging and enrichment.

Goal types' tendency to map to certain strategies suggests a relationship between prey and how a predator chooses to forage next. IFT already specifies that how a person forages depends on the contents of the predator's current patch and how they assess value and cost in that patch and that assessment is in part, based on their current prey. But, the relationship between the goal types and strategies suggests that certain foraging strategies are pursued for certain types of prey. This relationship may provide explanations for how programmers estimate scent based on their types of information that they are looking for. For example, a programmer foraging for at 4-groups goal type appears to have a higher expected value per expected cost for enrichment strategies than between-patch strategies. Thus, if the goal type of the programmer can be determined, the estimates of their expected values and cost can be more accurately estimated.

4.4.3 Unifying Diet with Strategies

We envision that similar approaches to the ones that we used in this chapter can serve to unify others works on programmers' diet (e.g., [Fritz & Murphy, 2010; Ko et al., 2006; LaToza & Myers, 2011; Sillito et al., 2006]) and foraging strategies (e.g., [Grigoreanu et al., 2010, 2012; Murphy et al., 2008; Pirolli & Card, 1999]). For example, LaToza and Myers' work which showed that programmers asked reachability questions [LaToza & Myers, 2010] could similarly be linked to Murphy et al.'s work on debugging strategies used by novices [Murphy et al., 2008]. RQ3's results showed that such a link already exists, with IFT providing the necessary abstractions to understand the relationship between diet and strategies.

With an understanding of the way diet and strategies connect, tool developers can leverage findings on how programmers behave and better support these behaviors in their tools. For example, looking at Table 4.10, when participants were pursuing a repeat patterns, they more often relied on a combination of within-patch foraging and enrichment. Two possible interpretations of the results follow. If the programmer was successful in their foraging, then this suggests that any tools supporting this type of diet should support both these types of foraging. If the programmer was unsuccessful in this type of diet, then the types of foraging may be indicative of what makes pursuing this diet difficult. Without unification, such insights would remain hidden.

4.5 Summary of Results

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

RQ1 (*what's*):

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

RQ2 (*how's*):

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

RQ3 (*what's meets how's*):

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

As Evans and Card summarize from Simon: "For an information system to be useful, it must reduce the *net* demand on its users' attention" [Evans & Card, 2008b]. Later, in Chapter 6, we will consider the essence of how programmers try to optimize this attention.

Chapter 5 Motivations and IFT for Software Engineering

In the previous two chapters we investigated two ways in which IFT can unify SE research. Here we step back and ask if IFT alone is enough to unify SE research, or if we need to consider other theories of human behavior. Thus, in this chapter, we investigate the role that Minimalist Learning Theory (MLT) [Carroll, 1998] may play in how programmers navigate during debugging.

The large number of goals reported in the previous chapter may be symptomatic of programmers' preference towards strategies that produce results quickly over those that require taking the extra time to learn. In fact, a prior observational field study of work practices revealed that "wherever possible, developers seem to prefer strategies that avoid comprehension" of existing code [Roehm et al., 2012]. Specifically, that study found that developers frequently tried to move forward with coding as quickly as possible, 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 [Brandt et al., 2009; LaToza et al., 2006; Maalej et al., 2014].

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" [Maalej et al., 2014].

This tendency of programmers to view learning as a costly task detracting from their efficiency is called *production bias* in Carroll's theory [Carroll, 1998; Carroll & Rosson, 1987]. To date, the effects of production bias on programmers have not been investigated in detail.

Production bias suggests that even subtle differences in what is motivating a programmer to be looking at code may influence how they forage. Carroll's theory [Carroll, 1998; Carroll & Rosson, 1987] would predict that a programmer who is trying to fix a defect will bias their foraging such that they maximize production. Let us consider two programmers working on the same defect, one who is fixing a defect (fixer), and another who is trying to understand the same defect's code (learner) "enough" to help someone else fix it. Does the fixer forage similarly to the learner as suggested by IFT (because they both have similar prey), or does their foraging differ, suggesting that IFT must consider the importance of motivation?

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). This chapter fills that gap by uncovering both *whether* and *how* production bias's tension of learning vs. efficiency affects information foraging during software maintenance. Specifically, we investigate the effects of production bias on foraging. Our investigation is grounded both in Carroll's theory, and in Information Foraging Theory.

We performed our investigation by conducting two qualitative laboratory studies across two environments. In both studies, one group of programmers 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 subtly different tasks to reveal and analyze differences in their behaviors from simultaneously an IFT and a production bias perspective.

In the first study, where participants worked in a desktop environment, we 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 programmers seek?
- RQ2 *(information patches)*: How does trying to fix a bug versus trying to learn about the bug affect *where in the environment* programmers 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* programmers 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 programmers use in making their foraging decisions?

In the second study, we repeat parts of the first study to investigate if its findings about Fix versus Learn participants generalize to a new environment. Our research questions repeat RQ2 and RQ3 but in a mobile environment instead of a desktop one:

- RQ5a (*patch types*): Does trying to fix a bug versus trying to learn about the bug affect *the where in the environment* that Mobile programmers forage from? If so, how?
- RQ5b (*cue types*): Does trying to fix a bug versus trying to learn about the bug affect the *types of cues* programmers attend to when navigating? If so, how?

5.1 Background and Related Work

5.1.1 Minimalist Learning Theory

In situations where people are primarily motivated to finish a task, information seeking is not the main activity, but a necessity for completing the task. These types of situations are well described by Minimalist Learning Theory (MLT) [Carroll, 1998]. MLT explains the learning motivations and behavior of *active users*, people in situations where learning is motivated by their current task. According to MLT, active users' focus on throughput (finishing their task as quickly as possible) leads to two conflicts that are mutually reinforcing. The first, a motivational paradox named *production bias*, describes how active users' focus on throughput reduces their motivation to spend time learning about the task they are working on, even if doing so would help them complete their tasks more effectively. The second, a cognitive paradox named *assimilation bias*, explains how active users apply what they already know to new situations and how this knowledge might lead to incorrect conclusions or misinformation regarding new knowledge [Carroll

& Rosson, 1987]. Carroll and Rosson argued that together these paradoxes amplify each other's effects.

Not only do programmers fit the role of the active user, but the two biases outlined above have implications for IFT, specifically for describing the *reactivity* that we have observed in previous studies [Lawrance et al., 2010; Piorkowski et al., 2012, 2013] (Chapter 3), where programmers suddenly changed their goals and their future navigations through the code. Consider the example of a programmer who encounters some piece of code that they decide would be too costly relative to the value that the code provides. Perhaps the code relies on an API that the programmer is unfamiliar with so it would distract them from their main task (production bias, high expected cost) or the API calls seem similar to ones that they have already seen elsewhere, but in fact are not (assimilation bias, low expected value). In both cases, the value of that code would increase if they later discover that that information was necessary to do their task. This would cause the programmer to return to the code and explore related code that would correspond to programmer behavior we have seen in prior studies. Intuitively, MLT provides a means for further understanding and explaining programmers' navigation while debugging. Therefore, a central goal of this chapter is to understand whether incorporating production bias as a factor is worth doing.

5.2 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.

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.

5.2.1 Procedure

The code that both groups worked with was from the jEdit project. All participants received a copy of bug report #3223[5] from jEdit's issue-tracking system[6]. 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.

5.2.2 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

[5] Bug #3223 is identical to bug #2548764 in Chapter 3 and Chapter 4. The id number changed because jEdit's developers migrated to a new repository.

[6] See Chapter 3 for the bug's details.

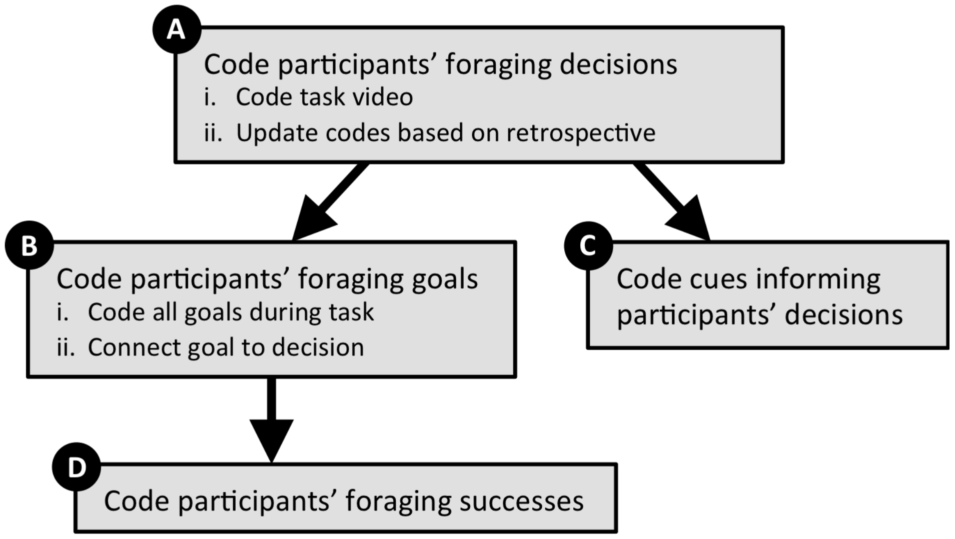

A
Code participants' foraging decisions
i. Code task video
ii. Update codes based on retrospective
B
Code participants' foraging goals
i. Code all goals during task
ii. Connect goal to decision
C
Code cues informing participants' decisions
D
Code participants' foraging successes

Figure 5.1. Our multi-phase qualitative coding technique.

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.

5.2.3 Qualitative Analysis Methods

We used a multi-phase qualitative coding process to analyze participants' information foraging behavior, as depicted in Figure 5.1. 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.

Code Set A: Participants' Foraging Decisions (RQ1–4)

We first focused on participants' foraging decisions—the moments they explicitly chose one patch over another (Figure 5.1.A.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 (Figure 5.1.A.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.

Code set B: Participants' Information Goal Types (RQ1)

For each foraging decision, we then coded the participant's information goal type (Figure 5.1.B). To identify participants' information goals as they foraged, we coded participants' information goals using 44 previously documented questions programmers ask [Sillito et al., 2006] (Figure 5.1.B.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 [Grigoreanu et al., 2012; Lawrance et al., 2013; Piorkowski et al., 2012; Pirolli & Card, 2005]. We then mapped the goals to corresponding foraging decisions whenever participants said that a foraging decision was connected to a particular information goal (Figure 5.1.B.ii).

Code Set C: Participants' Cues (RQ3)

For each foraging decision, we also coded the types of cues the participant considered when making the decision (Figure 5.1.C). 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 5.3.3) for clarity of presentation.

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 fulfilled, as unsuccessful if the participant said it was not, or as unknown if the participant gave no indication (Figure 5.1.D).

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.

5.2.4 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 [Agresti & Kateri, 2011], 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).

5.3 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).

5.3.1 RQ1. What to Look For? Fixers' and Learners' 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.

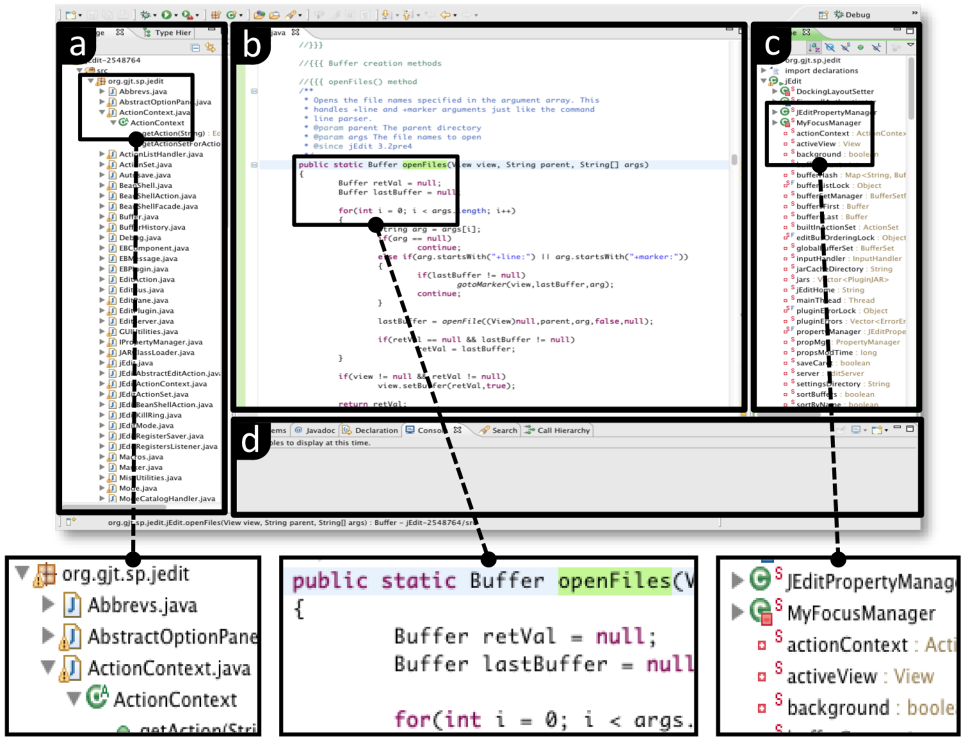
P11L: Okay I should be looking for this text (*delete lines*). How do I find this text?

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.

5.3.2 RQ2. Turning Points: The Patches from which Fixers and Learners 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 occurred.

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)



```
//}}}

//{{{ Buffer creation methods

//{{{ openFiles() method
/**
 * Opens the file names specified in the argument array. This
 * handles +line and +marker arguments just like the command
 * line parser.
 * @param parent The parent directory
 * @param args The file names to open
 * @since jEdit 3.2pre4
 */
public static Buffer openFiles(View view, String parent, String[] args)
{
	Buffer retVal = null;
	Buffer lastBuffer = null;

	for(int i = 0; i < args.length; i++)
	{
		String arg = args[i];
		if(arg == null)
			continue;
		else if(arg.startsWith("+line:") || arg.startsWith("+marker:"))
		{
			if(lastBuffer != null)
				gotoMarker(view,lastBuffer,arg);
			continue;
		}

		lastBuffer = openFile((View)null,parent,arg,false,null);

		if(retVal == null && lastBuffer != null)
			retVal = lastBuffer;
	}

	if(view != null && retVal != null)
		view.setBuffer(retVal,true);

	return retVal;
```

Figure 5.2. An information environment (Eclipse) as a programmer (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.

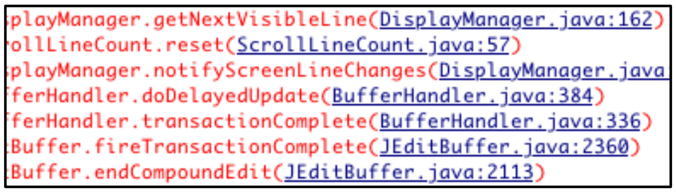
was a *patch type*, containing one or more patches. For example, Figure 5.2 depicts an Eclipse window with three patch types, each containing at least one patch.

Table 5.1 shows the types of patches and characterizes the content each type offers. For example, Package Explorer patches (e.g., Figure 5.2.a) give high-level structural overviews of the code, whereas Stack Trace patches (e.g., Figure 5.3) 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., Figure 5.4) give a list of navigable search results. Table 5.1 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.) Patch types may also exist elswhere, but we did not include them in Table 5.1.

With these patch types, we analyzed participants' navigation decisions (Code Set A in Section 5.2.3) among the patches. Because each navigation decision by a forager had a

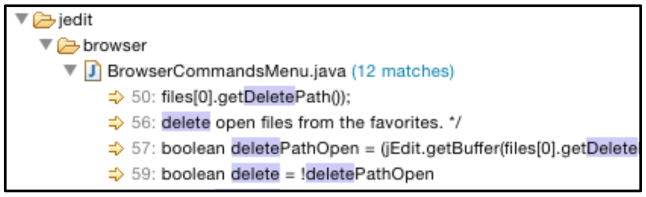
Patch Type	Information and Navigational Links	#
Editor	Provides a listing of the code in a file. Identifiers in the code are linked to associated Call Hierarchy and/or Search Results patches. Example in Fig. 1b.	37
Stack Trace	Provides a list of code locations along an execution path that produced an 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
Total		106

Table 5.1. Types of patches in which participants made foraging decisions. The rightmost column is the number of foraging decisions made from that patch type.



```
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)
```

Figure 5.3. Example Stack Trace patch.



```
▼ 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
```

Figure 5.4. Example Search Results patch.

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 (Figure 5.2.a), and make a navigation decision, clicking on one of the hyperlinked items in the view. As a result, the Editor (Figure 5.2.b) 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 Figure 5.5, Fix and Learn participants demonstrated significant differences in the *starting* patch types, i.e., those *from* which they made navigation decisions (Analysis of deviance, $\chi^2(11)=28.8$, $p=0.002$).

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., Figure 5.2.a and Figure 5.4, 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

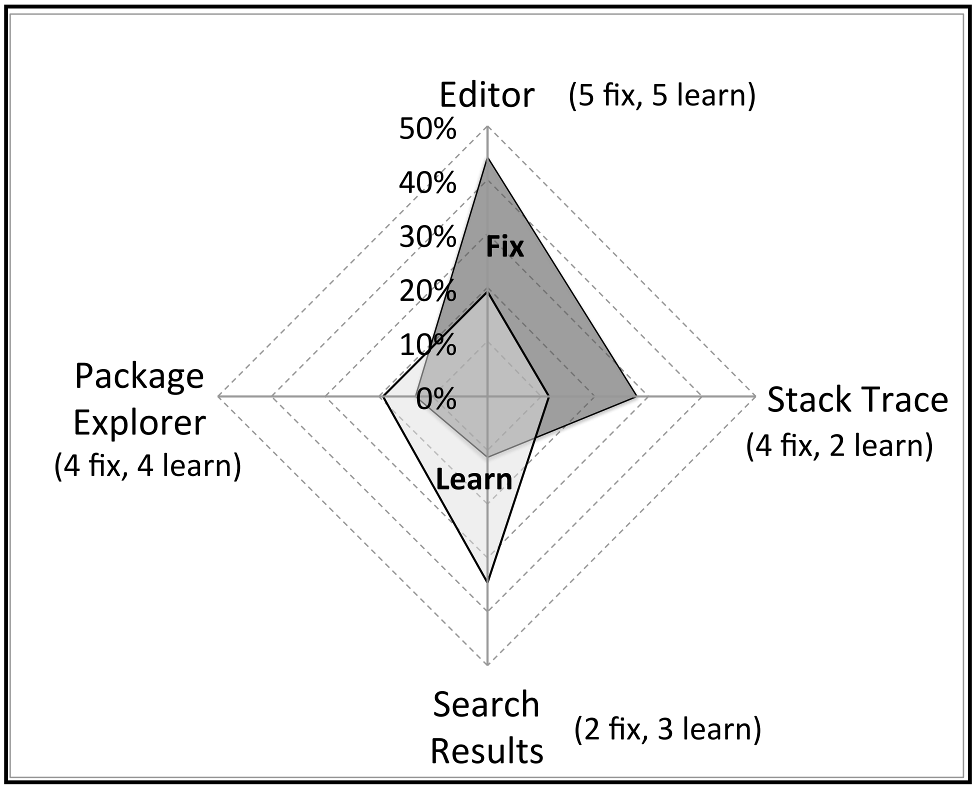


Figure 5.5. The proportion of patches where each treatment's participants made foraging decisions (showing the 4-most often used patch types). Percentages indicate the number of navigation decisions in a patch type divided by the total number of navigation decisions (per treatment). In parentheses is the number of participants in each treatment that made at least one navigation decision in that patch type.

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., Figure 5.3). 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.

For example, consider two participants: Learn Participant P5L and Fix Participant P3F. P5L foraged extensively within a ToC-like Search Results patch. As Figure 5.6 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 Figure 5.7 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.

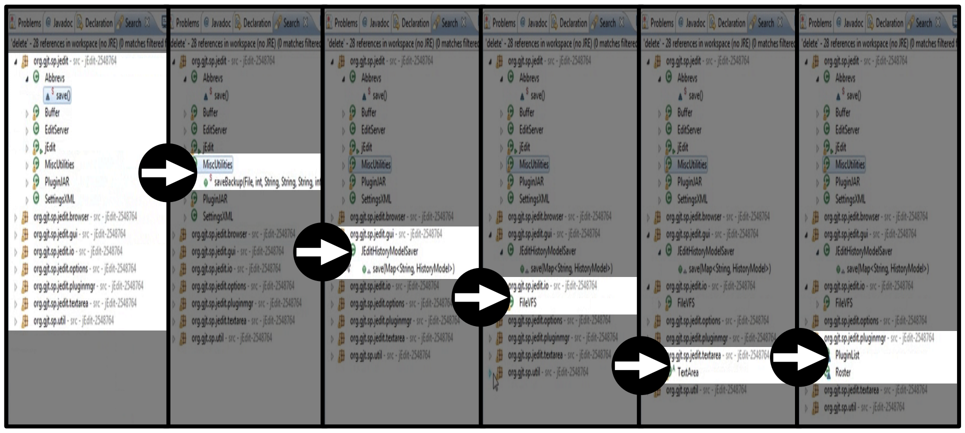
P3F in fact commented explicitly on a desire to avoid certain types of patches…

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.

5.3.3 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 programmer has seen often inspires their ensuing decisions. Thus, for RQ3, the cue types we coded (Table 5.2) 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.

For example, as Figure 5.8 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



Problems
Javadoc
Declaration
Search
'delete' - 28 references in workspace (no JRE) (0 matches filtered
org.gjt.sp.jedit - src - jEdit-2548764
Abbrevs
save()
Buffer
EditServer
jEdit
MiscUtilities
saveBackup(File, int, String, String, String, int
PluginJAR
SettingsXML
org.gjt.sp.jedit.browser - src - jEdit-2548764
org.gjt.sp.jedit.gui - src - jEdit-2548764
JEditHistoryModelSaver
save(Map<String, HistoryModel>)
org.gjt.sp.jedit.io - src - jEdit-2548764
FileVFS
org.gjt.sp.jedit.options - src - jEdit-2548764
org.gjt.sp.jedit.pluginmgr - src - jEdit-2548764
PluginList
Roster
org.gjt.sp.jedit.textarea - src - jEdit-2548764
TextArea
org.gjt.sp.util - src - jEdit-2548764

Figure 5.6. 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.

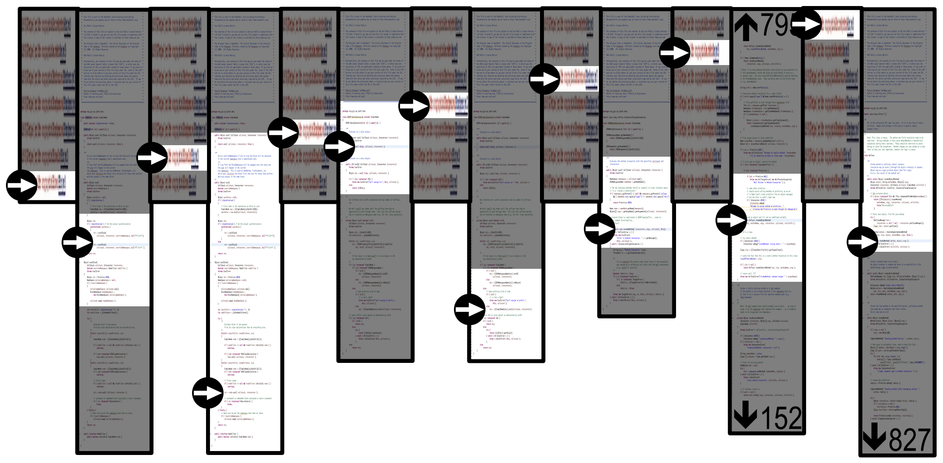
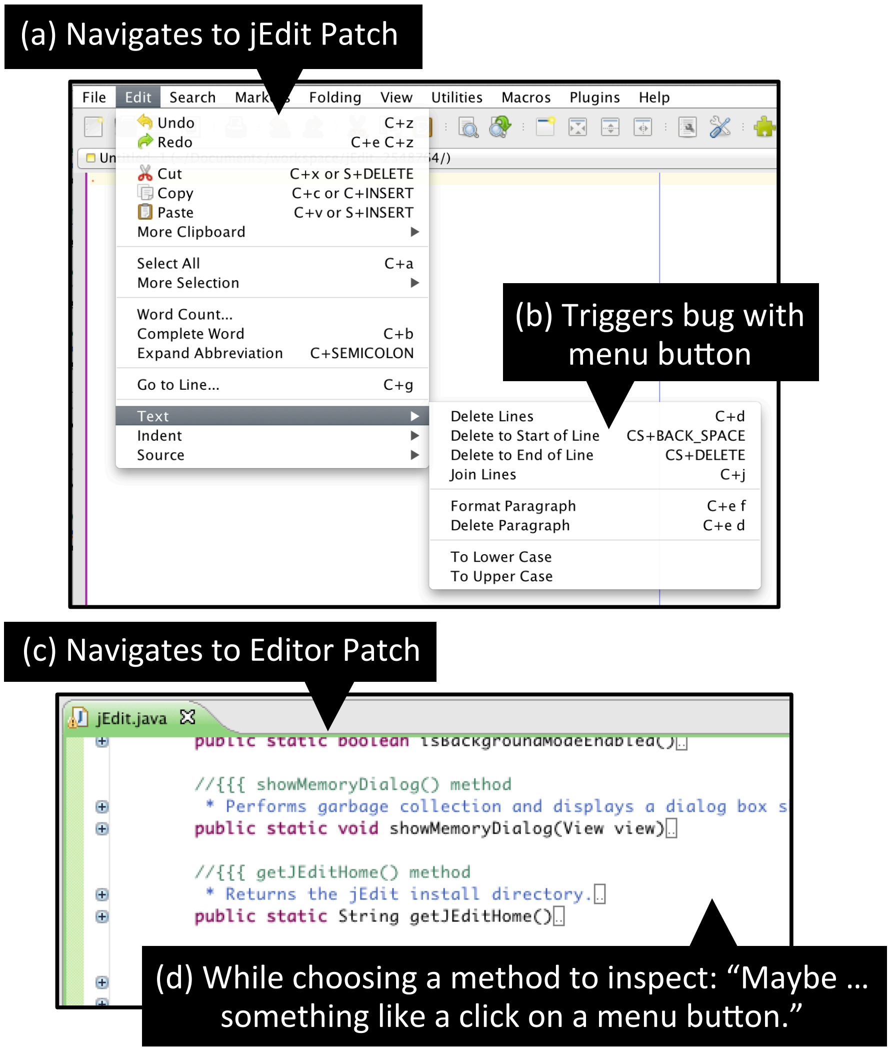


Figure 5.7. Fix Participant P3F used the Stack Trace patch from Figure 5.3 as an index to forage for particular lines of code in a variety of Editor patches. Each box from left to right shows a snapshot of a patch, alternating between the Stack Trace patch and Editor patches. The arrows show the links 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).

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
Total		144

Table 5.2. The cue types to which participants attended when making foraging decisions. The rightmost column is the number of foraging decisions in which participants attended to that cue type.


(a) Navigates to jEdit Patch
File Edit Search Markers Folding View Utilities Macros Plugins Help
Undo C+z
Redo C+e C+z
Cut C+x or S+DELETE
Copy C+c or C+INSERT
Paste C+v or S+INSERT
More Clipboard
Select All C+a
More Selection
Word Count...
Complete Word C+b
Expand Abbreviation C+SEMICOLON
Go to Line... C+g
Text
Indent
Source
Delete Lines C+d
Delete to Start of Line CS+BACK_SPACE
Delete to End of Line CS+DELETE
Join Lines C+j
Format Paragraph C+e f
Delete Paragraph C+e d
To Lower Case
To Upper Case
(b) Triggers bug with menu button
(c) Navigates to Editor Patch
jEdit.java
(d) While choosing a method to inspect: "Maybe … something like a click on a menu button."

```
public static boolean isBackgroundModeEnabled()

//{{{ showMemoryDialog() method
 * Performs garbage collection and displays a dialog box s
public static void showMemoryDialog(View view)

//{{{ getJEditHome() method
 * Returns the jEdit install directory.
public static String getJEditHome()
```

Figure 5.8. 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).

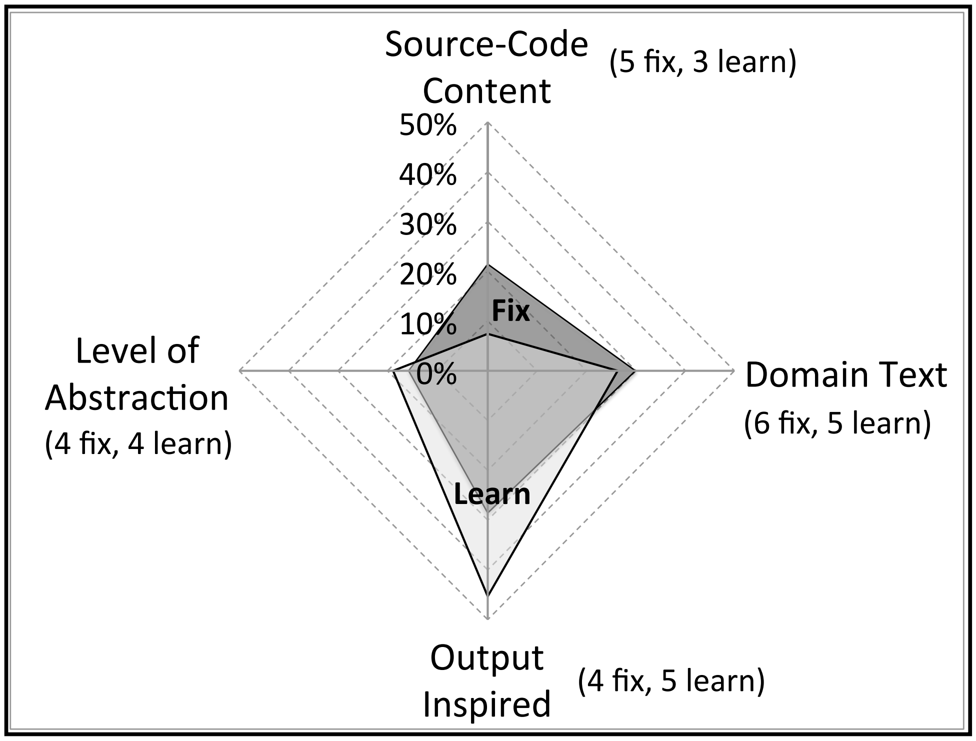


Figure 5.9. The proportion of cue types each treatment's participants attended to at each navigation decision, showing the four most-often attended cue types. Percentages indicate the number of cue types attended divided by the total number of cue types attended to (per treatment). In parentheses is 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.

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, $\chi^2(19)=33.3$, $p=0.02$). Figure 5.9 highlights these differences for the most-attended cue types.

One difference apparent in Figure 5.9 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 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.

Fix participants' foraging stayed "closer to the code" than that of Learn participants in two additional ways, as well. First, as shown in Figure 5.9, 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.

5.3.4 RQ4. How Should I Go About Foraging? Fixers' and Leaners' 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 5.3, "Most-Used Tactic" 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, $\chi^2(3)=5.93$, $p=0.12$).

Treatment	Participant	Most-Used Tactic	Success Rate Favorites	Success Rate 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%)

Table 5.3. Participant usage and success rates for the Favorites and Switching tactics.

However, a difference between treatments becomes apparent with situations where the outcome of a foraging action was *successful* (code set D in Section 5.2.3). 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, $\chi^2(1)=5.49$, $p=0.019$).

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.

5.4 Generalizing Fix Versus Learn to Mobile Environments

So far, we have investigated the role of production bias in foraging during debugging in a single environment using a single defect. However, theories need to be evaluated over several different situations and domains. We take a first step in this direction by repeating the study in a new environment, so as to investigate whether the same differences between Fix and Learn participants generalize to the mobile environment.

As mobile devices become ever more pervasive and ever more powerful, the likelihood of programming directly on a mobile device as opposed to through a traditional computer seems ever more likely. Being able to develop mobile applications directly on mobile devices tightens the traditional edit-compile-run cycle (compared to develop with an emulator or a mobile device tethered to a desktop) thereby reducing the cognitive gap between program code and execution. For example, according to a study performed by Nguyen, et al., programmer productivity is significantly enhanced when programmers are offered the opportunity to write mobile applications directly on mobile devices, particularly when writing small applications [Nguyen et al., 2012b].

In this section, we investigate if our findings generalize to the mobile environment. We focus on RQ2 and RQ3: patch types that participants make navigations[7] from and the cues types that they attended to. (In this section, we will refer to the previous results on Eclipse and jEdit as the *Desktop* environment. This section's environment will be referred to as the *Mobile* environment.) Thus our research questions are:

- RQ5a (*patch types*): Does trying to fix a bug versus trying to learn about the bug affect *the patch types* that Mobile programmers forage from? If so, how?
- RQ5b (*cue types*): Does trying to fix a bug versus trying to learn about the bug affect the *types of cues* programmers attend to when navigating? If so, how?

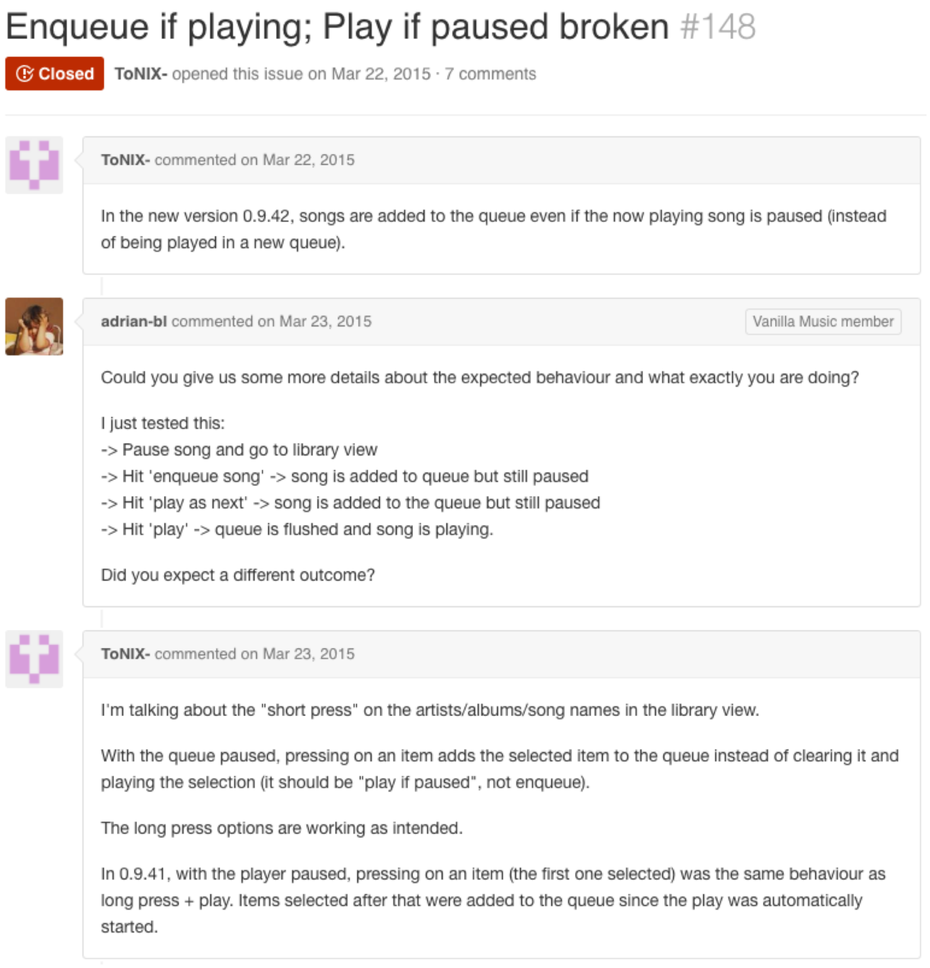
5.4.1 Methodology and Analysis

Mobile Environment Study Design

The methodology follows a similar approach as to the one in Section 5.2.3, with some modifications to account for the mobile environment we used: an IDE that runs natively on Android mobile devices called AIDE[8].

[7] Only 7% of navigations were foraging decisions for Mobile participants compared to 28% on desktop. Due to the lack of data, we analyzed the full set of 501 desktop navigations and 217 mobile navigations.

[8] http://www.android-ide.com/


Enqueue if playing; Play if paused broken #148

Closed ToNIX- opened this issue on Mar 22, 2015 · 7 comments

ToNIX- commented on Mar 22, 2015

In the new version 0.9.42, songs are added to the queue even if the now playing song is paused (instead of being played in a new queue).

adrian-bl commented on Mar 23, 2015 Vanilla Music member

Could you give us some more details about the expected behaviour and what exactly you are doing?

I just tested this:
-> Pause song and go to library view
-> Hit 'enqueue song' -> song is added to queue but still paused
-> Hit 'play as next' -> song is added to the queue but still paused
-> Hit 'play' -> queue is flushed and song is playing.

Did you expect a different outcome?

ToNIX- commented on Mar 23, 2015

I'm talking about the "short press" on the artists/albums/song names in the library view.

With the queue paused, pressing on an item adds the selected item to the queue instead of clearing it and playing the selection (it should be "play if paused", not enqueue).

The long press options are working as intended.

In 0.9.41, with the player paused, pressing on an item (the first one selected) was the same behaviour as long press + play. Items selected after that were added to the queue since the play was automatically started.

Figure 5.10. Vanilla Music's bug #148.

We randomly divided our participants into two treatments: Fix and Learn, giving them the same prompt to each treatment as described before. Participants from both treatments worked on the same Android app and defect for the study. The application was Vanilla Music[9], a mature open-source music player for Android. Vanilla Music is written in Java and it contains 67 classes and 13,369 non- comment lines of code. We tasked participants to work with issue #148[10] taken from the Vanilla Music's issue tracking system on GitHub. The issue described a problem with playback regarding an enqueued song and is presented in Figure 5.10.

[9] Vanilla Music is a popular application that has between 500,000 and 1,000,000 downloads on the Google Play Store.
[10] https://github.com/vanilla-music/vanilla/issues/148

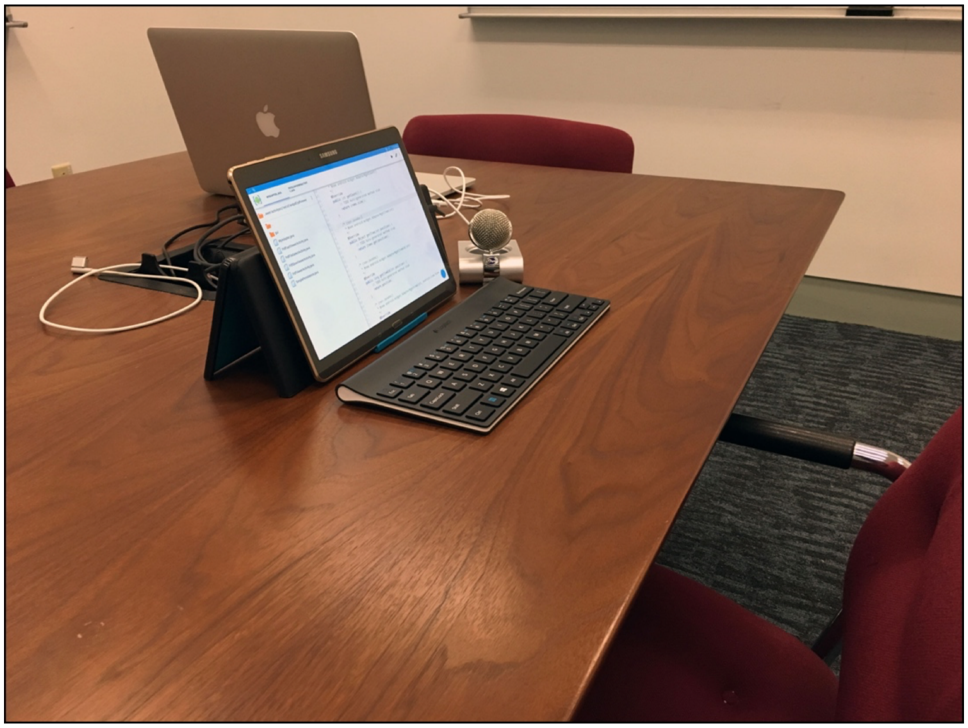
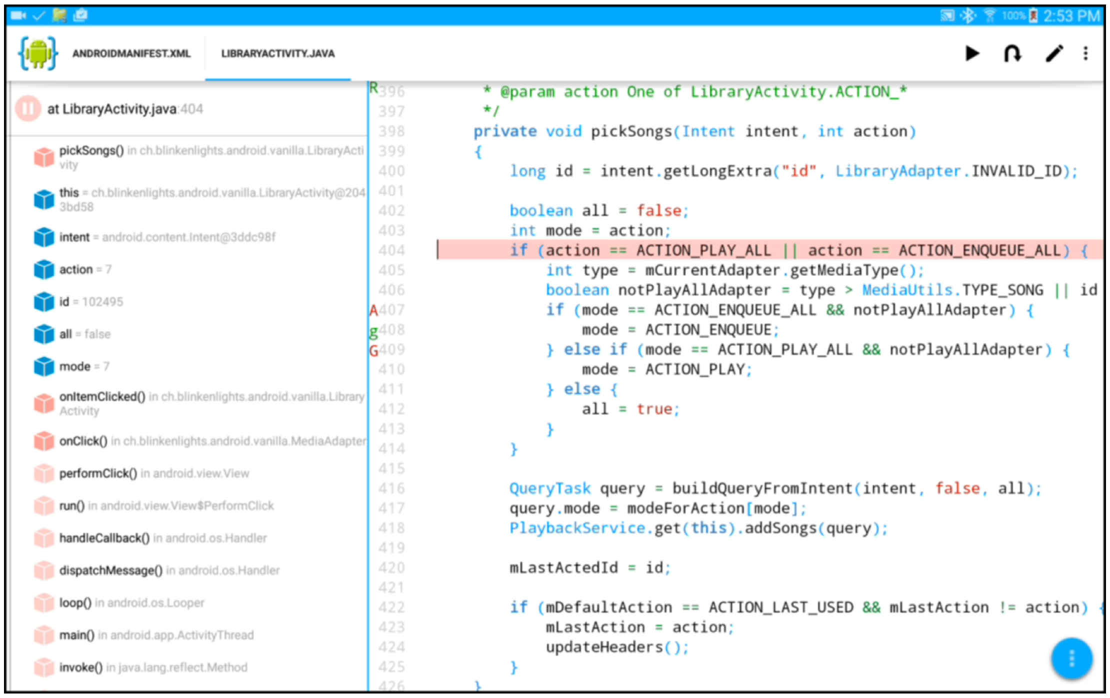


Figure 5.11. The lab study setup. Participants used a combination of touch input on the tablet and typing on the keyboard during their tasks.

During the experiment, participants used a Samsung Galaxy Tab S 10.5-inch tablet paired with a Bluetooth keyboard (Figure 5.11). This tablet has 3GB of RAM and a 1.9Ghz Quad-core + 1.3 GHz Quad-core processor, making it one of the most powerful Android tablets currently available. The programming environment was AIDE[11], a full-featured programming IDE for Android. Figure 5.12 shows a screenshot of AIDE interface.

Each session started with a brief introduction explaining the experiment followed by a brief background questionnaire. We then provided a 10-minute tutorial to teach AIDE's features to each participant. Features included basic navigation, various find utilities, setting up breakpoints, using the debugger, running the application and accessing the LogCat output. We allowed participants to ask any questions they had about using AIDE during the session as we did not want unfamiliarity with the IDE to impede participants from

[11] AIDE is one of the most popular Android IDE with 2 million downloads on the Google Play Store.



```
at LibraryActivity.java:404

pickSongs() in ch.blinkenlights.android.vanilla.LibraryActivity
this = ch.blinkenlights.android.vanilla.LibraryActivity@2043bd58
intent = android.content.Intent@3ddc98f
action = 7
id = 102495
all = false
mode = 7
onItemClicked() in ch.blinkenlights.android.vanilla.LibraryActivity
onClick() in ch.blinkenlights.android.vanilla.MediaAdapter
performClick() in android.view.View
run() in android.view.View$PerformClick
handleCallback() in android.os.Handler
dispatchMessage() in android.os.Handler
loop() in android.os.Looper
main() in android.app.ActivityThread
invoke() in java.lang.reflect.Method
```

```java
     * @param action One of LibraryActivity.ACTION_*
     */
    private void pickSongs(Intent intent, int action)
    {
        long id = intent.getLongExtra("id", LibraryAdapter.INVALID_ID);

        boolean all = false;
        int mode = action;
        if (action == ACTION_PLAY_ALL || action == ACTION_ENQUEUE_ALL) {
            int type = mCurrentAdapter.getMediaType();
            boolean notPlayAllAdapter = type > MediaUtils.TYPE_SONG || id
            if (mode == ACTION_ENQUEUE_ALL && notPlayAllAdapter) {
                mode = ACTION_ENQUEUE;
            } else if (mode == ACTION_PLAY_ALL && notPlayAllAdapter) {
                mode = ACTION_PLAY;
            } else {
                all = true;
            }
        }

        QueryTask query = buildQueryFromIntent(intent, false, all);
        query.mode = modeForAction[mode];
        PlaybackService.get(this).addSongs(query);

        mLastActedId = id;

        if (mDefaultAction == ACTION_LAST_USED && mLastAction != action)
            mLastAction = action;
            updateHeaders();
        }
    }
```

Figure 5.12. AIDE in its debugger mode. The left view can be changed to fit the context. Here it gives the stack trace and the currently initialized variables. Other contents include the File Explorer, Search Results, Current Breakpoints and the LogCat.

working. The tablet recorded the screen (and finger presses) along with the participants' utterances.

Each participant was placed into either the Fix or Learn treatment and given 30 minutes to complete the task. Following the task, we performed a retrospective interview where we played back the entire session to the participant and asked them questions about what they did. Both the treatment tasks and the retrospective followed the same procedure as described in Section 5.2. However, unlike before, we asked participants questions about *each navigation* instead of just the navigations where foraging decisions were made.

Participants

The participants consisted 8 professional programmers (7 males and 1 female) at a large software development company. They had 1 to 38 years (mean of 19.25 years) of software development experience, 2 to 20 years (mean 11.875 years) of Java development experience and they had 0.5 to 5 years (mean of 2 years) of Android development experience. None of the programmers were familiar with AIDE or Vanilla Music. Their ages ranged from 20 to 55 years.

Foraging Decisions versus Navigations

Due to a lack of foraging decisions in the Mobile environment, we instead decided to focus our analyses using the full set of navigation data for both Desktop and Mobile environments. Hence, our analysis is based on *navigations* instead of foraging decisions.

Unlike a desktop IDE, the text cursor's position in the file was not an accurate way to determine navigations, since participants often scrolled the editor's view instead of moving the text cursor. Instead, we counted a navigation 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. We used participant validation to verify each of these navigations during the retrospective, by asking participants if they were considering the content of the method during these hesitations.

Code Set: Patch Types

We coded the patch types using the methodology described in Section 5.2.3. AIDE had fewer patch types available than Eclipse. The patches and the number of instances that participants made a foraging decision from that patch are presented in Table 5.4.

Code set: Cue Types

We coded the cue types using the methodology described in Section 5.2.3. The cue types and the number of navigations that participants attended to a given cue types for a navigation are presented in Table 5.5.

Changes to Desktop Methodology

To make a fair comparison between Mobile (where we counted navigations) and Desktop (where we counted foraging decisions), we recoded all the patch types that Desktop participants navigated from to include all navigations. A summary of the counts for each patch type is presented in Table 5.6.

Patch Type	Information and Navigational Links	#
Editor	Provides a listing of the code in a file.	99
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).	64
Package Explorer	Provides a list of the files in the project. List items open the files in the Editor Patch.	47
Stack Trace	Provides a list of code locations along an execution path that produced an exception (i.e., internal error) in the running Vanilla Music program. List items are linked to the associated lines of code (opened in an Editor patch).	5
Debug	Provides a list of the currently executing code and program state for the user-specified breakpoint location.	2
Total		217

Table 5.4. The patch types that Mobile participants navigated from. The rightmost column is the number of occurrences for each patch type was navigated from.

Cue Type	Definition: Participants' utterances about…	#
Source-code Content Inspired	… cues related to source code they had seen, such as relating to a par- ticular variable or parameter, or reminiscent of a code comment	67
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"	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
File Type	… cues related to the type of a file they had seen, such as Java vs. XML	6
Output-Inspired	… cues related to Vanilla Music out- put they had seen, such as thrown exceptions (errors) or GUI widget labels	5
Documentation-Inspired	cues related to external documentation they had seen, such as the bug report	2
Total		155

Table 5.5. The cue types to which Mobile Participants attended to when navigating. The rightmost column is the number of navigations in which participants attended to that cue type.

Patch Type	# Desktop	# Desktop (no P5)	# Mobile
Debug	135	23	6
Editor	259	235	95
Open Call Hierarchy	14	14	n/a
Outline View	27	25	n/a
Package Explorer	33	31	47
Search Results	33	33	64
Stack Trace	74	74	5

Table 5.6. The total number of each patch type that participants made navigations from for both Desktop and Mobile participants. The column labeled '# Desktop (no P5)' has total with Participant 5 removed because he was an outlier.

The table presents two columns for Desktop participants to reflect an outlier in the navigation data. Desktop Participant 5's behavior was non-typical as he was not familiar with how to use Eclipse's debugger. This became evident when early in his debugging task, he performed a Google search to find "how to set the current line of execution." After viewing four posts on Stack Overflow[12], and expressed frustration about not finding a solution. He said, "Maybe what I'm trying to do can't be done in Eclipse, or I just don't know how to do it." Despite the lack of understanding, Desktop Participant 5 decided to use the debugger regardless. When using the debugger, he rapidly stepped through code, often not pausing to read what was on the screen (for 112 navigations). This resulted in him having a much higher number of navigations than other participants. Alone, he accounted for 140 out of 553 navigations across 11 participants. Desktop Participant 3 made the second largest number of navigations with 61, less than half of Desktop Participant 5's 140 navigations. Thus, the following present the data for Desktop participants without Desktop Participant 5.

5.4.2 Results RQ5a: Patch Types

Where Fixers and Learners Navigated From on Mobile

Fixers and Learners significantly differed in which patch types they chose to navigated from (Analysis of deviance, $\chi^2(4)=28.49$, $p<0.001$) as shown in Figure 5.13. The largest

[12] Stack Overflow is a website where programmers can post questions and reply to posted questions (stackoverflow.com).

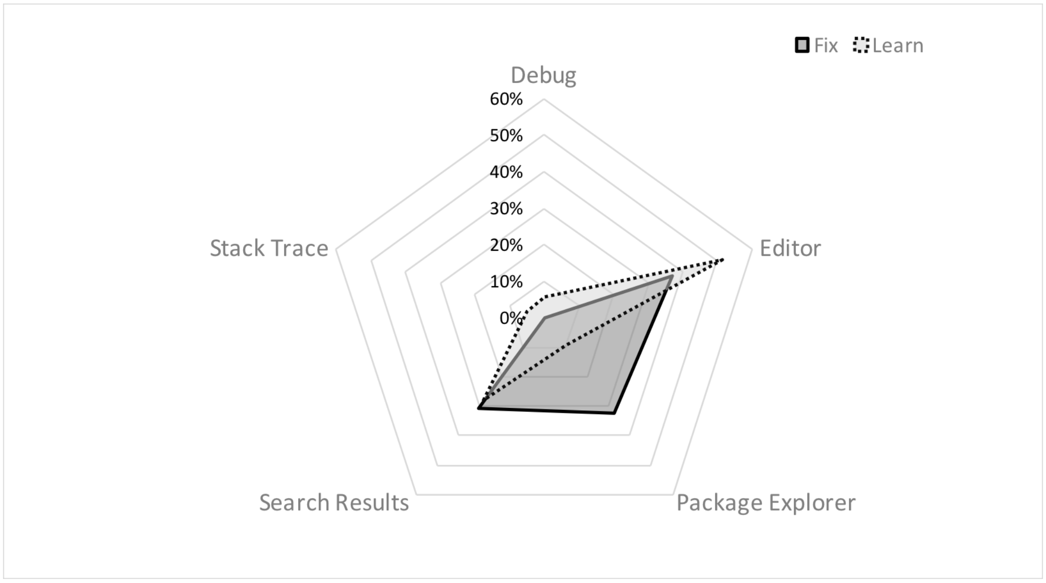


Figure 5.13. The proportion of patch types where each treatment's Mobile participants made navigations from. Percentages indicate the number of navigations in a patch type divided by the total number of navigations (per treatment). We limited the chart to include the 5 most commonly patch types navigated from.

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.

Patch Types Across Environments

Although the result that Fixers and Learners foraged differently generalized across the Desktop and Mobile environments, their most common patch types did not. Figure 5.14 shows the five most commonly navigated from patch types for Fixers on Desktop and Mobile. The graphs show very little overlap suggesting that Fixers on Desktop did not emphasize the same patch types as Fixers on Mobile did.

Mobile learners likewise emphasized different patch types than their Desktop counterparts. Figure 5.15 shows the five patch types Desktop versus Mobile Learn participants most often navigated from. Of these, the only one in common was the Editor, which was similarly emphasized by both Desktop (54%, 101 out of 186 navigations) and Mobile Learners (52%, 53 out of 103 navigations).

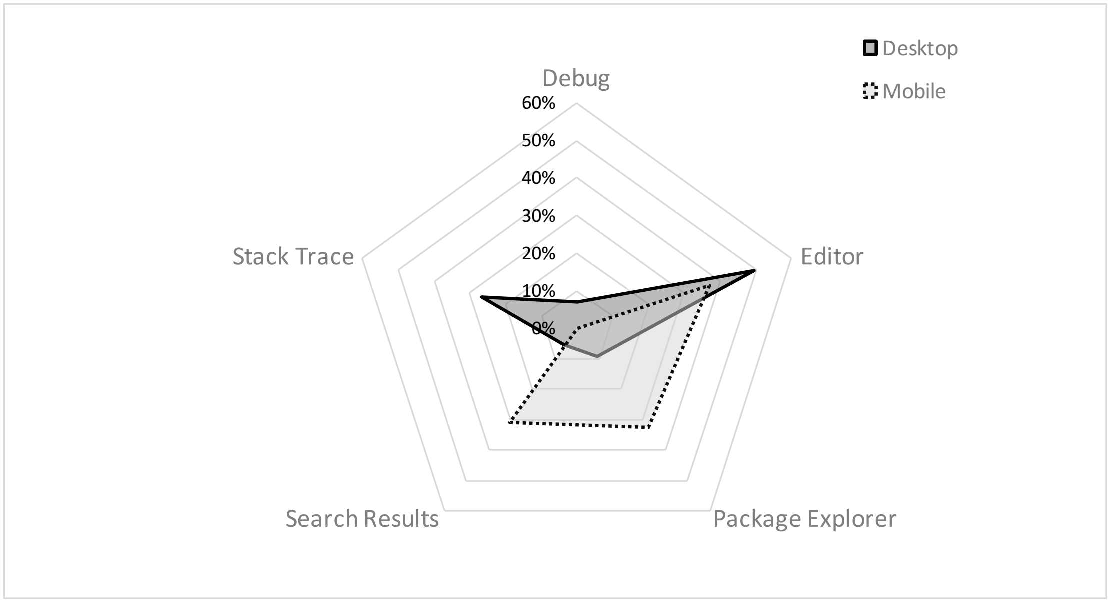


Figure 5.14. The proportion of patch types where each environments' Fix participants made navigations from. Percentages indicate the number of navigations in a patch type divided by the total number of navigations (per environment's Fix treatment). We limited the chart to include the 5 most commonly patches navigated from

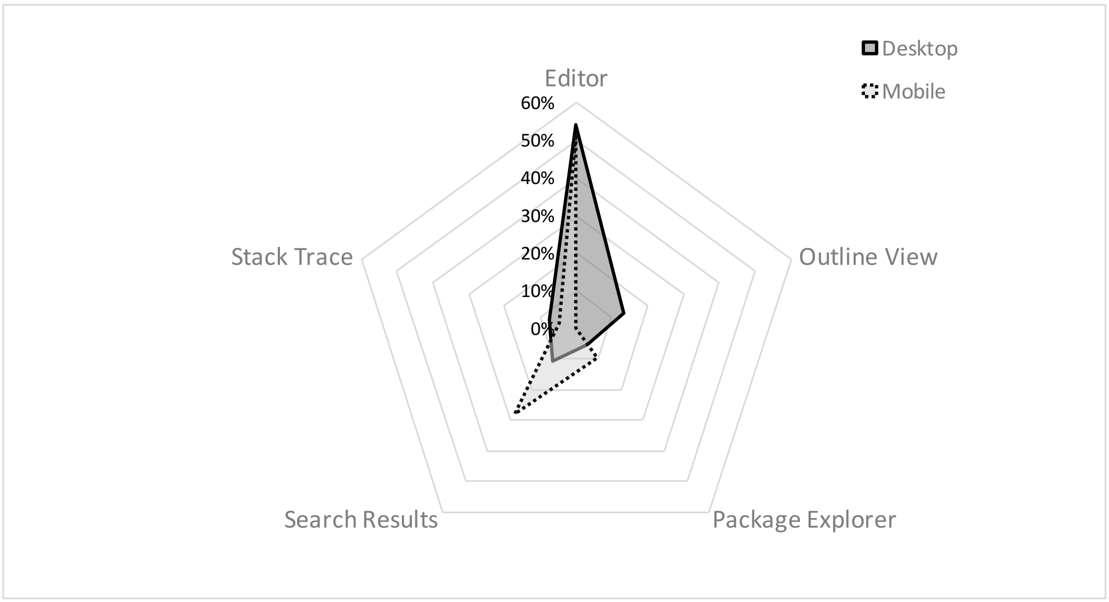


Figure 5.15. The proportion of patch types where each environment's Learn participants made navigations from. Percentages indicate the number of navigations in a patch type divided by the total number of navigations (per treatment per environment). We limited the chart to include the 5 most commonly patch types navigated from.

Reasons we cannot dismiss the differences between Desktop and Mobile Participants	Fixers' patch types affected	Learners' patch types affected
Insufficient navigation data on some patch types	Debug, Package Explorer, Stack Trace	Debug
Patch type did not exist in Mobile	Outline View, Open Call Hierarchy	
Participants verbalizations did not explain why	Search Results	Editor, Package Explorer, Search Results
Differences in the bugs		Stack Trace: For Desktop, most Stack Trace navigations came from an exception stack trace that printed every time participants reproduced the bug. No such stack trace was printed when Vanilla Music's bug was reproduced.

Table 5.7. A summary of potential confounds preventing conclusions about Mobile Fixers versus Desktop Fixers or Mobile Learners versus Desktop Learners in terms of patch types.

Even viewed at a higher level, participants' patch type emphases were different between the two environments. Recall our observation that on Desktop, Fixers tended to navigate from patch types that were low-level (index-like) and close to the code whereas Learners tended to navigate from ones that were high-level (ToC-like). On Mobile, Fixers and Learners did not have the same preferences. Fixers instead split their patch types choices between both low-level patch types such as Editor and Search Results[13], as well as the higher-level, hierarchy-revealing patch type, the Package Explorer. In contrast, Learners almost exclusively pursued low-level patch types like the Editor and Search Results.

Why Desktop Versus Mobile Emphasized Different Patch Types

For the patch types that differed between Desktop and Mobile, we considered several reasons for the differences between environments that we cannot dismiss. As Table 5.7 summarizes, the possibilities range from lack of data, differences in the bugs, and differences in the IDE.

[13] Unlike Eclipse, the Search Results patch type in AIDE does not reveal hierarchy and is therefore, a low-level patch type.

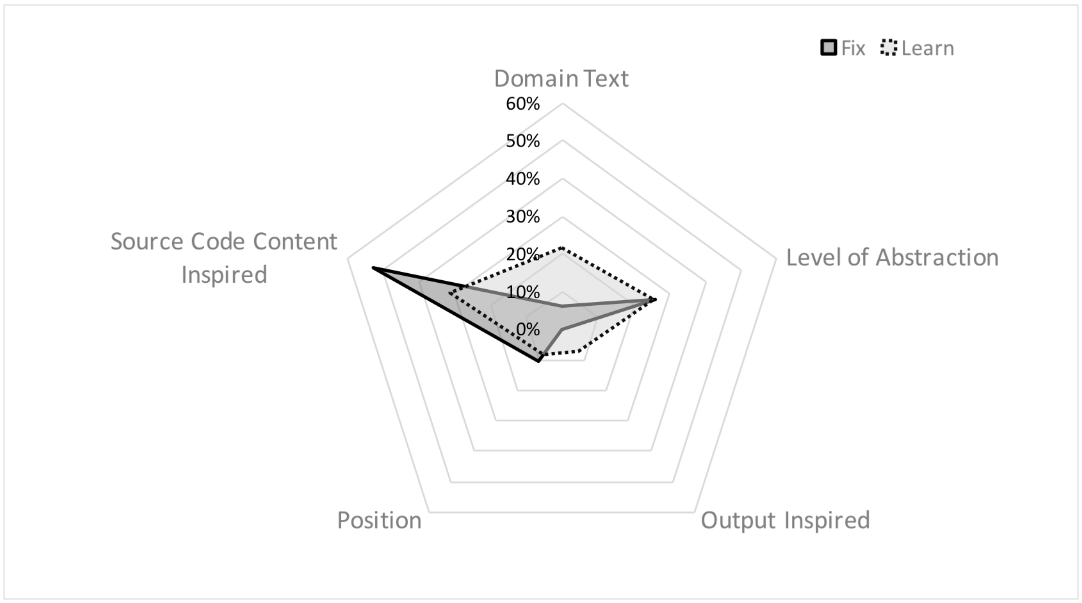


Figure 5.16. The proportion of cue types attended to by Mobile participants for each treatment. Percentages indicate the number of navigations where those cues were attended to divided by the total number of navigations (per treatment). We limited the chart to include the 5 most commonly attended to cue types.

However, these potential confounds are present only in comparing Fixers versus Fixers or Learners versus Learners; there are no such confounds in comparing Fixers versus Learners within one environment. Thus, the fact that Fixers versus Learners forage differently does generalize, but the question of *which* patch types Fixers versus Learners emphasize remains open.

5.4.3 Results RQ5b: Cue Types

Which Cues Fixers and Learners Attended To

Participants also differed significantly in terms of the cue types that they attended to when making navigations (Analysis of deviance, $\chi^2(6)=19.73$, $p=0.003$) as shown in Figure 5.16. 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.

Cue Types Across Environments

As with patch types, even though the results that Fixers and Learners foraged differently in terms of cue types generalized across Desktop and Mobile *which* cue types Fixers versus Learners emphasized differed in the two environments. Figure 5.17 shows the five most often attended to cue types for Fixers in both environments. The figure shows little overlap, suggesting that Desktop Fixers and Mobile Fixers emphasized different cue types in their foraging.

Mobile Learners also emphasized different cue types compared to Desktop Learners. Figure 5.17 Figure 5.18 shows the five most often attended cues for Desktop versus Mobile Learners. Only one cue type was similarly attended to. Desktop Learners attended to the Level of Abstraction cue type for 25% of the coded cue types (24 out of 98) and Mobile Learners attended to 26% of them (18 out of 70).

Why Desktop Versus Mobile Emphasized Different Cue Types

Like patch types, the differences between Desktop and Mobile Learners had several potential confounding factors, or were otherwise difficult to explain. Table 5.8 summarizes. In addition to the reasons provided in the patch type section above, differences between the jEdit bug and the Vanilla music UI provided additional confounding factors.

Like patch types, these confounds are present only when comparing a treatment across two environments (Desktop Fixers versus Mobile Fixers or Desktop Learners versus Mobile Learners), not when comparing within one environment (Mobile Fixers to Mobile Learners). Thus, the differences in cue types between Fixers and Learners generalizes. However, due to the confounding factors, the question of *which* cue types Fixers versus Learners emphasize remains open.

5.5 Discussion

5.5.1 Production Bias

Production bias affected how Fixers and Learners perceived scent as shown by their differing preferences in both patch types and cue types. Recall that the patch types that

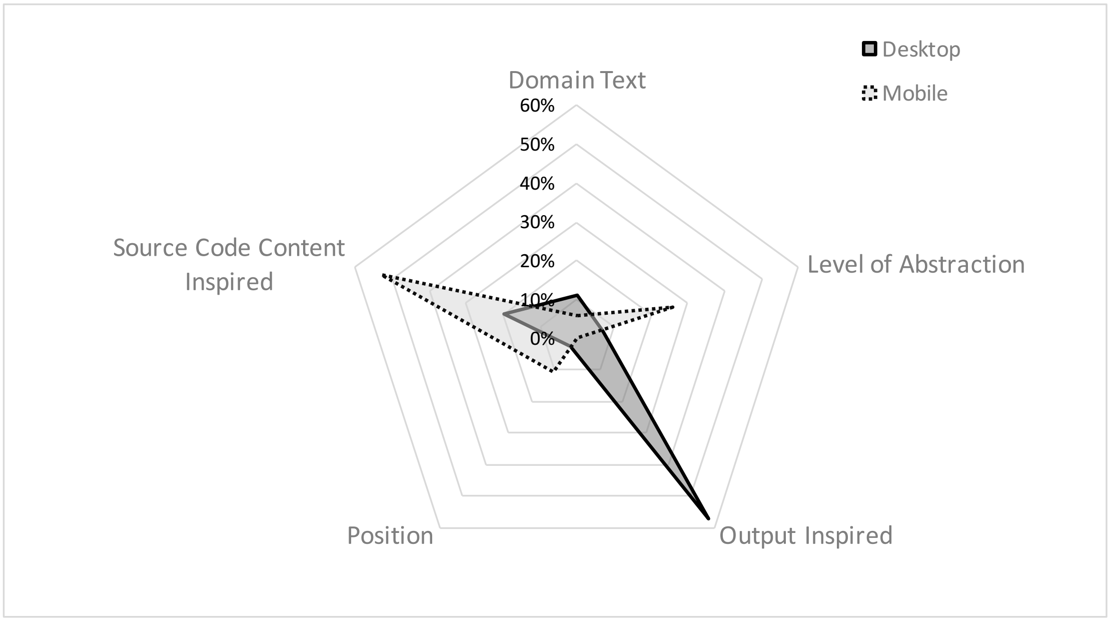


Figure 5.17. The proportion of cue types attended to by Fix participants for each environment. Percentages indicate the number of navigations where those cues were attended to divided by the total number of navigations (per environment's Fix Treatment). We limited the chart to include the 5 most commonly attended to cue types.

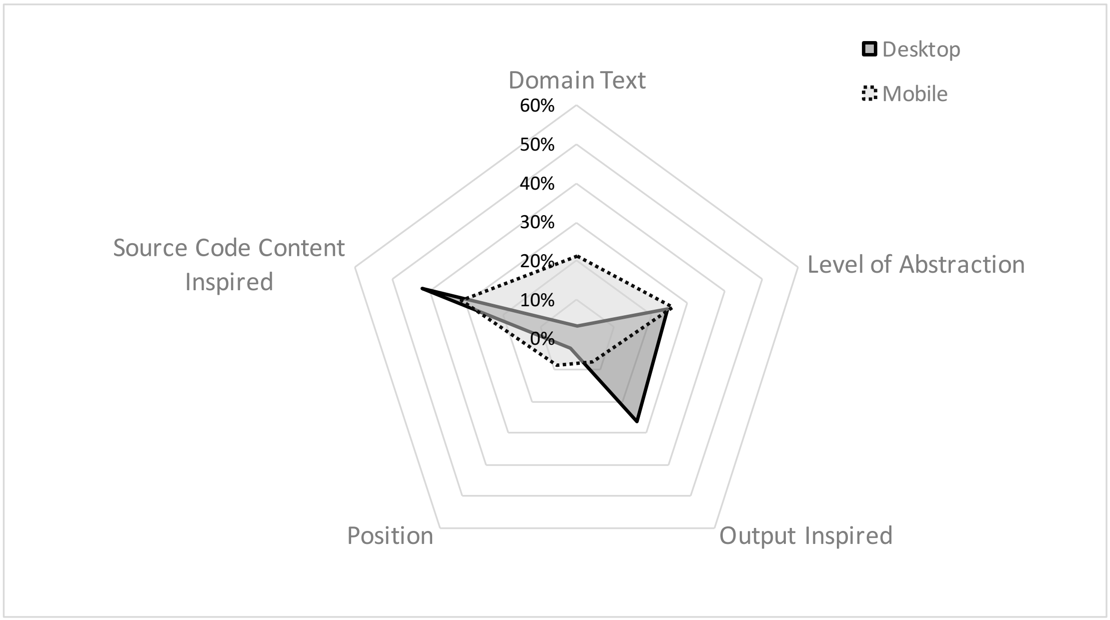


Figure 5.18. The proportion of cue types attended to by Learn participants for each environment. Percentages indicate the number of navigations where those cues were attended to divided by the total number of navigations (per environment's Learn Treatment). We limited the chart to include the 5 most commonly attended to cue types.

Reasons we cannot dismiss the differences between Desktop and Mobile Participants	Fixers' cue types affected	Learners' cue types affected
Insufficient navigation data on some cue types	Documentation, Familiarity, File Types, Source Code Appearance Inspired	Documentation, Familiarity, File Type, Position, Source Code Appearance Inspired
Differences in the bugs	Output Inspired: For Desktop, Output Inspired cues came from an exception stack trace that appeared whenever the bug was reproduced. No such stack trace appeared when Vanilla Music's bug was reproduced.	
Differences in the defective program's GUI	Domain Text: For Desktop participants, the GUI provided several words relevant to the defect as part of the UI (e.g. "Delete Line(s)") in the menu). For Mobile participants, the UI was dependent on the media content: Vanilla Music's UI provided artist names and track titles, instead of words related to the bug.	
Participants' verbalizations did not explain why	Position, Source Code Content Inspired	Source Code Content Inspired

Table 5.8. A summary of potential confounds preventing conclusions about Mobile Fixers versus Desktop Fixers or Mobile Learners versus Desktop Learners in terms of cue types.

Desktop Learners preferred tended to include cues that revealed where in the code's hierarchy a given patch was (i.e. search results that were categorized by package), and they took advantage of these patch types to quickly navigate between patches within that patch. In contrast, Desktop Fixers tended to stay in one patch at a time, focusing on the details of that patch.

These differences may be explained as differences in Structure scent (Section 3.6.4) between Fixers and Learners, with Desktop Learners more attuned to structure cues than Desktop Fixers. In the case of Mobile Fixers and Learners, the importance of Structure scent appears to have been reversed with Mobile Fixers more focused on ToC-like patch types than Mobile Learners.

We speculate that, because of differences in cost in the two environments, the way that participants in each environment perceived scent differed as well. Consider the case of running the application in a debugger. In the case of Eclipse, the application opens and can bel placed next to the debugger. Thus, the programmer can easily see changes in the application as she steps through the debugger. In AIDE, using the debugger is much costlier because only one application can be on the screen at a time. Thus, as the programmer steps through the code in AIDE, they do not see how the application changes, instead having to switch back and forth, remembering the previous visual state and identifying the changes in the new visual state.

These differences in cost may have been magnified for Mobile participants since none had ever used AIDE. Therefore, their expectations from using other IDEs may have hindered their ability to use the Mobile IDE. Fundamentally, these mismatches of expectation may be explained by assimilation bias, but as of now remain an open question.

5.5.2 Assimilation Bias

This chapter investigated the effect of production bias on how programmers forage during debugging, but the implications MLT's other bias—assimilation bias—remain unexplored. Recall that assimilation bias occurs when active users apply what they already know to new situations potentially leading to incorrect conclusions or misinformation regarding new knowledge. For example, an experienced text-editor programmer may already have an expectation of how a text editor program is structured. If her expectation aligned with jEdit's structure, that may be beneficial. But if their prior experiences with text editor code used a different paradigm, assimilation bias may manifest in their foraging.

Going one step further, Carroll argued that production bias and assimilation bias were mutually reinforcing [Carroll, 1998]. The interplay between biases may cause the experienced text editor programmer to be so convinced that their assumed structure is correct. This could cause the programmer to ignore evidence to the contrary just to make progress on her task, a clear influence on her foraging behavior. More work is needed on these types of question and to understand assimilation bias's effects on foraging.

5.5.3 Cue Types in IFT

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., [Lawrance et al., 2008b; Piorkowski et al., 2012]). 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.

5.5.4 Cue Types May Explain Differences Between Mobile and Desktop

Recall that mobile fixers and learners attended to different cue types and patch types than desktop fixers and learners. One possible reason for the differences is the value and costs associated with similar cues in each environment. For example, search results on the Desktop are presented in a hierarchical view, providing a better estimate of value (by providing context) and reducing actual cost (by enabling the programmer to read the line of code without navigating there). In contrast, the search results on the Mobile environment are presented as shortened text, due to the smaller screen size, and do not provide any additional context. This makes it more difficult for the programmer to estimate the value of the patch and also increase the actual cost (relative to Desktop) since they are likely to have to navigate to the patch just to determine if the search result was relevant. Such differences would alter how programmers perceive scent in the environment.

5.6 Summary of Results

The results of our empirical studies show, for the first time, how Minimalist Learning Theory's concept of production bias can influence programmers' foraging for information. Programmers 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 on both the Desktop and Mobile environments:

- RQ2 (*Patch Types*): Learn vs. Fix participants' turning points—should I go here or should I go there—occurred in significantly different types of patches. On Desktop, 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.

- RQ3 (*Cue Types*): Desktop 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.
- RQ4 (*Foraging Tactics*): Desktop Learn participants' successful tactics were different from those of Fix participants. Desktop 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.
- RQ5 (*Generalizability*): As with the Desktop environment, Mobile participants emphasized significantly different patch types and cue types, generalizing the result that Fixers forage significantly differently from Learners regardless of environment. However, the question of which patch types and cue types Fixers versus Learners emphasize remains open.

A recurring theme discussed in the previous chapters is the role of scent as a unifying construct. In Chapter 3, navigational factors leveraged by SE tools corresponded to types of scent and explained how programmers navigated. In Chapter 4, scent served as a possible explanation for the relationship between goals and strategies. Here, in Chapter 5, how programmers perceived scent may explain the differences we observed in programmers' foraging between the two environments.

Given this seeming importance to software engineering tools, the fundamentals of scent as applied to SE tools are understudied. We address this gap in the next chapter.

Chapter 6 Scent Fundamentals: Value and Cost as SE Unifiers

Despite software engineering researchers developing numerous tools to reduce programmers' foraging costs (e.g., [DeLine et al., 2005b; Henley & Fleming, 2014; Karrer et al., 2011; Kevic et al., 2014; Krämer et al., 2013; Majid & Robillard, 2005; Piorkowski et al., 2012; Singer et al., 2005]), little is known at the foundational level of programmer navigation—how well programmers go about *choosing* where to navigate. Thus, in this chapter, we address this gap and investigate one of the fundamental aspects of navigation as explained by IFT: programmers' expectations of *value and cost*.

Recall that programmers are not omniscient, that is, they cannot exactly determine the actual value and cost of a navigation prior to making the navigation. Instead, they rely on cues in the environment to predict the value and cost of the navigation. When they carry out these navigation decisions, they may be in for disappointments if (1) their destination does not provide as much value as expected or (2) the cost of extracting the information or getting to it is higher than expected.

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

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

6.1 Background and Related Work

Recall from Chapter 2 that IFT's central proposition says that the predator treats foraging as an optimization problem. More specifically, the predator's foraging actions try to maximize the value V of information gain from a patch per their cost C of getting to and interacting with that patch, i.e.:

$$\text{Predator's desired choice} = max\frac{V}{C}$$

However, predators' knowledge is imperfect, so they make their choices based on their *expectations* of value V and of cost C, i.e.:

$$\text{Predator's selected choice} = max\frac{E(V)}{E(C)}$$

How accurate are foragers in forming these expectations? Software engineering research has much to say about information professional programmers seek (e.g., [Lawrance et al., 2013; Sillito et al., 2006]), but has not systematically considered the question of how well programmers can *predict* these values and costs, or the accompanying implications for SE tools. These are the questions this chapter investigates.

Little empirical work has investigated the specific question of programmers' abilities to predict the value and cost of their navigations. One study examined navigations of analysts through documents and code as they attempted to recover requirement traceability between use cases and Java classes, revealing that the subjects spent a disproportionate amount of time in low-value patches (i.e., more time than the value of those patches justified) [Niu et al., 2013]. This result was consistent with an earlier study into the challenges of information foraging by programmers, such as the fact that searches in code frequently failed to turn up desired results, and that programmers spent substantial time organizing and re-organizing files and bookmarks in the IDE [Ko et al., 2006]. Another study of how programmers search online resources corroborated that searches often lead to irrelevant code and supporting documents, except after programmers had determined the right search terms for specific subtopics [Bajracharya & Lopes, 2012]. Our work builds on these results.

6.2 Empirical Study Methodology

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

6.2.1 Participants, Procedures, and Task

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

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

The participants' task was to debug an actual bug (issue #3223) in the jEdit open source project[14]. Participants used the Eclipse IDE on a Windows PC to complete the

[14] See Chapter 3 for the bug's details.

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

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

Figure 6.1. Retrospective semi-structured interview questions.

task. We also allowed them to use any other tools they wanted to complete the task as they saw fit, including using the web.

After the debugging session, we conducted a retrospective semi-structured interview. The purpose was to collect participants' expected value before a navigation, and then the value they actually received. To collect the data, we played the session video for the participant, pausing it at each *to-method* event and *away-from-method* event on the video, to ask the questions shown in Figure 6.1. The to-method pauses occurred just as the participant navigated to a Java method (just before arriving there), and the away-from-method pauses occurred just as the participant navigated away from the method (just before seeing the new location). If participants visited other kinds of files (e.g., properties files), we asked them the same questions as for the methods.

6.2.2 Qualitative Analysis

To analyze the data, we used a qualitative coding approach to map key concepts ("codes") to participants' navigations [Seaman, 1999]. Specifically, we coded the videos (which included both navigation actions and corresponding verbalizations by the participants), whenever participants talked about the value or cost of a navigation.

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

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

6.3 Results

6.3.1 RQ1: Programmers' Expectations of Value

Did Programmers Get the Value They Expected?

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

To perform this measurement, we coded participants' responses to the retrospective interview questions (Figure 6.1). For *expected value* (before they processed the patch), we coded their responses to the "to-method" questions, and to measure their perceived *actual value* of the method (after they processed it), we coded their responses to the "away-from-method" questions. Table 6.1 shows the code set we used to analyze these navigations.

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

Table 6.1. Code set for expected (prior to navigating) and actual (after navigating) values.

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

Table 6.2. Participants' expectations of value vs. actual value. Gray cells highlight navigations in which participants had some degree of disappointment (50.8% of navigations).

By these measures, the participants' expectations of the information value they would receive for their foraging efforts were optimistic: they expected Necessary or Sufficient information from about 84% of their navigations (Table 6.2's top two rows' totals). The first row shows navigations in which participants expected to find *everything* they needed (Necessary and Sufficient: about 25%), and the second shows navigations in which they expected to find at least *something* they needed (Necessary, but not Sufficient: about 59%).

However, many of these expectations of value were not fulfilled. As the Table 6.2's bottom row shows, 63 of participants' 179 navigations (about 35%) produced lower value than expected. Adding to these disappointments, 28 of the 29 "desperation" navigations (not expected to be either Necessary or Sufficient)—in which participants actually expected no value but tried anyway—indeed led to no actual value. Thus, in total, about 51% of participants' navigations (highlighted cells in Table 6.2) ended in some degree of disappointment in the information value they received.

Participants rarely found *more* value than they expected. As Table 6.2's first column shows, participants found higher information value than expected in only 5 navigations, and only one was a participant "lucking into" information in a desperation navigation.

Why: The Challenges of Signposting

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

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

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

False Advertising: Content + Where the Cue Points

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

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

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

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

P8 (when asked if he was hoping for something):

"some more connection to deletion of the actual text...[but] it was just the abstraction"

The Problem with Synonym Cues

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

P2: "'clear.bsh', is that related to deleting? No it's not"

P7: "I'm assuming 'invalidate' means 'delete' ... Uh, it just doesn't delete"

We were surprised to see the problems that arose with synonyms as cues, because several tools use synonyms directly or indirectly to good effect (e.g., tools powered by natural language vocabulary devices like TF-IDF). For example, the search tool FindConcept uses synonyms to expand the search query [Shepherd et al., 2007; Sridhara et al., 2008], and Krec uses standard English synonyms [Robillard & Chhetri, 2015].

These approaches bring to mind seminal work on what was originally termed the "vocabulary problem" [Furnas et al., 1987]. That paper showed how huge variations in designers' terminology across numerous application domains are an inherent property of the English language. This result suggests not only the advantage of automatically agglomerating synonyms but also its disadvantage—bringing together synonym-related patches greatly expands programmers' search space, as with P2, P6, and P8 above. Thus, too little synonym agglomeration produces too many false negatives, but too much synonym agglomeration produces too many false positives.

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

The synonyms problem may relate to Ge et al.'s observation that over 90% of relevant synonyms are unique to software engineering [Ge & Murphy-Hill, 2014]. For example, in software, "invoke" is a synonym of "execute", and "instantiate" is a synonym of "create". They point out that tools could use a thesaurus tailored to the lexicon of SE. Our results are consistent with this point, but also suggest that the problems with synonym cues may extend beyond that solution.

Cues That Answered the "Wrong" Question

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

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

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

Table 6.3. *E(V)* research challenges with value estimation and the participants who experienced them.

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

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

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

Some systems try to address "where does" problems. For example, WhyLine [Ko & Myers, 2008] is well-suited for "where does" questions about state and variables, and Reacher [LaToza & Myers, 2011] answers "where does" questions about methods. However, proof-of-concept tools like these need to be investigated in the context of the entire IDE. Such tools require programmers to navigate away from the "main" part of the environment into other tools and screens, potentially causing them to lose context and adding to programmers' costs simply by the cost of navigating to other tools.

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

An open problem: The "Value Estimation" problem with programmers' navigations

Table 6.3 summarizes the research challenges in better supporting programmers' attempts to predict patch values before paying the cost of navigating to those patches. These research challenges come together to reveal a large, open problem space:

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

The challenges identified so far in this chapter show that this problem is nuanced, difficult, and multidimensional. Even so, addressing this problem promises high rewards. Recall from Table 6.2 that solving this problem could potentially improve programmers' navigation efficiency by up to 51%.

6.3.2 RQ2: Programmers' Expectations of Cost

Did Participants Incur the Costs They Expected?

Participants did not verbalize their expectations of cost before navigating so we did not measure $E(C)$ and C separately. Instead, we measured how $E(C)$ related to C, because after navigating they often verbalized a navigation's cost exceeding their expectations ($C > E(C)$). Thus, our code set (Table 6.4) allocated these verbalizations among the two possible ways costs can be incurred: by navigating between patches (C_b), or by processing within the patch once there (C_w). Thus, $C = C_b + C_w$. The results in Table 6.4 show that participants discussed facing unexpected costs in 66 of the 179 navigations analyzed (36.9%).

Why: Unanticipated Costs Between Patches

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

P1: "Uh, the whole thing was really frustrating. The code was hard to read."

P5: "looking for ... but then I got so lost in [that method], that I didn't really fully understand what was going on."

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

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

Table 6.4. Frequency of actual costs (C_b or C_w) that were unexpectedly higher than the programmers had expected ($E(C_b)$ or $E(C_w)$). (The total is 66 instead of 80 because categories can co-occur in the same navigation.

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

Prey in Pieces, Scattered among Multiple Patches

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

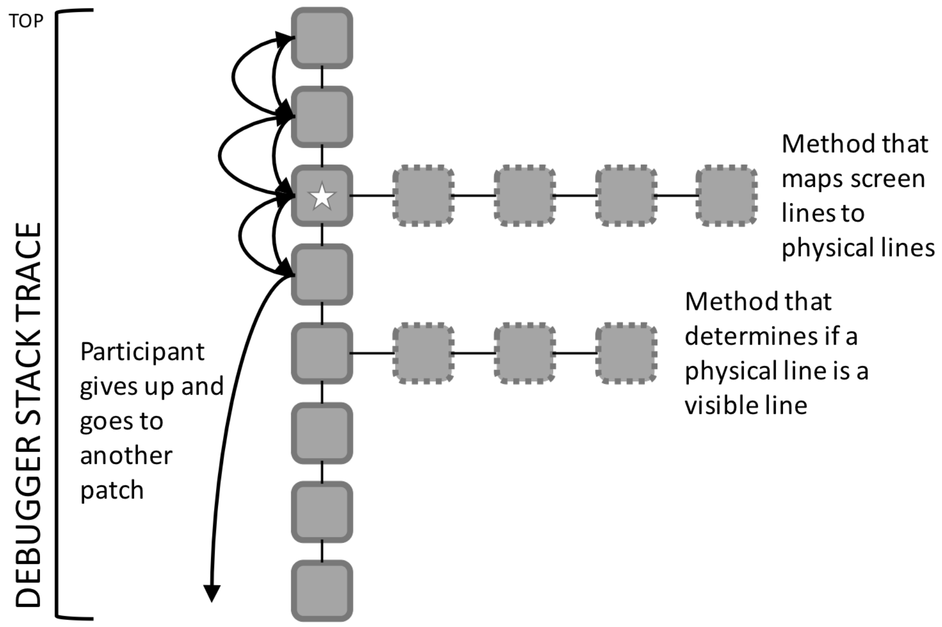

TOP
DEBUGGER STACK TRACE
Method that maps screen lines to physical lines
Method that determines if a physical line is a visible line
Participant gives up and goes to another patch

Figure 6.2. P1 was looking for the relationship between screen lines and visible lines, after seeing both in the starred method. But jEdit has three line types, so he would have *also* needed to understand physical lines from the dashed locations (far right).

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

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

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

To illustrate the cost involved in establishing such context, consider P1’s case. As Figure 6.2 shows, for P1 to build the context he wanted, he would have had to not only move up the call stack to find the relevant relationships and documentation, but also had to locate and navigate through the call relationships through the dashed methods in the figure before putting together his desired prey. Of course, P1 had no way of knowing this, and after foraging within the first four methods of the call stack, he gave up.

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

The Path to the Prey Is Too Long, with No End in Sight

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

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

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

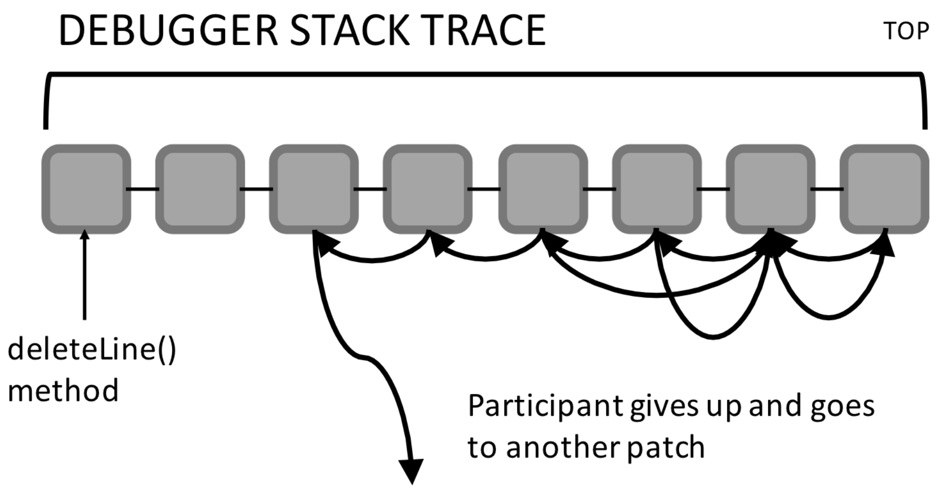


Figure 6.3. P3 navigated down the debugger's stack frames searching for methods related to folding or deletion. After several navigations down the stack, he gave up only three methods away from the method he was looking for.

For example, when P3 was looking for methods related to folding or deleting text, he set a breakpoint in the exception-throwing method that he identified earlier. He then foraged through the sequence of methods in the debugger's stack frames working his way down the stack, sometimes returning to a previous frame to regain lost context. After several navigations, he gave up—still three methods away from the `deleteLine` method that he was looking for (Figure 6.3). P1, P2, P3, P4, P5, P8, and P9 all experienced this expense of navigating through long sequences of patches en route to their desired prey.

Sometimes There is No Navigable Path

Some participants could not find a path to their prey because the information they wanted was located in a different topology altogether. A *topology* is a collection of patches and the links between them. In this study, one topology was the code itself, with units of code (such as methods or classes) being the patches and the ways to navigate between them (like scrolling or using various IDE navigation affordances) being the links. Another topology, disjoint from the code topology, was the jEdit running instance, with its own patches and navigation affordances not connected to code. In the Eclipse IDE, participants sometimes formulated their foraging goal while in one topology, but had to

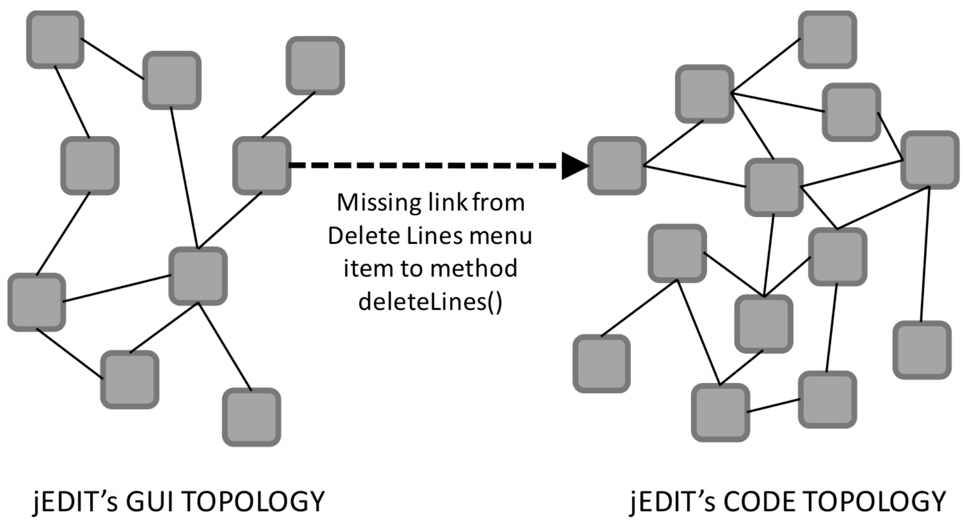

Missing link from Delete Lines menu item to method deleteLines()
jEDIT's GUI TOPOLOGY
jEDIT's CODE TOPOLOGY

Figure 6.4. P7 said he wanted to navigate from the Delete Lines menu action to the deleteLines method in the code, but the environment had no link from the action to the code it triggered. (Solid lines: links present. Dashed line: missing link.)

fulfill the goal in another. What was missing was a way for participants to easily navigate between related patches from one topology to another.

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

This inability to move between topologies in a low-cost way primarily manifested when participants were mapping runtime GUI behavior to code. For example, several participants, while recreating the bug in jEdit's running instance, formulated the goal of finding the code that was triggered by GUI actions. However, after formulating that goal, they then had to switch over to jEdit's code and start a fresh set of navigations, since there was no direct way to go from executing the action in jEdit to the code that handles that action. Instead, participants located the relevant code by using search tools, by investigating the stack trace, or by setting a breakpoint and stepping through code. Figure 6.4 shows one common missing link between the two topologies, mapping the Delete Lines menu action to the `deleteLine` method.

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

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

There are a few beginnings toward addressing the disjoint topologies challenge. For example, Whyline [Ko et al., 2006] builds a path just-in-time to bridge the gap between two topologies: from GUI output to its relevant source code, and SketchLink [Baltes et al., 2014] links sketches to code. Mining approaches like Chen and Grundy's [Chen & Grundy, 2011] are also emerging to find relationships among disjoint topologies such as documentation and source code. However, Whyline does not scale to a program of jEdit's size, approaches like Chen/Grundy's do not support navigations per se, and few of the approaches we have located reason with more than two disjoint topologies. Still, these beginnings provide promising starts upon which to build.

An Open Problem: The "Cost Estimation" Problem with Programmers' Navigations

As Table 6.4 showed, about 37% of programmers' costs were much higher than they had expected. In essence, programmers had to navigate to patches without knowing what it would cost until *after* they had paid—a situation not unlike writing a blank check for the coffeemaker of our earlier analogy. Table 6.5's summary of cost-related foraging re-

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

Table 6.5. Research challenges for cost estimation and the participants who faced them.

search challenges contributing to these issues reveal a substantive and difficult open problem space analogous to the Value Estimation Problem presented in Section 6.3.1:

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

6.4 RQ3: Literature Analysis

To answer our third research question, whether recent trends in software engineering research have begun to address these problems, we conducted a literature analysis of 302 papers from three literature repositories. The first repository was the 99 papers cited in the most recent (2013) journal paper surveying SE tools that contribute to programmers' information foraging [Fleming et al., 2013], which included, for example, tools helping collect information for debugging, reuse, or infering what a programmer seeks, and for recommending appropriate resources (e.g., [Cottrell et al., 2008; Ko & Myers, 2008; Robillard & Chhetri, 2015; Sawadsky et al., 2013]). The 2013 journal paper [Fleming et al., 2013] sampled literature from a wide range of dates, so to ensure currency, we added two very recent repositories. Thus, the second repository was FSE'14 (104 papers), which was the most recent FSE available at the time we began this analysis, and the third was ICSE'14 (99 papers), i.e., the same year as the FSE repository.

6.4.1 Analysis Methodology

From the resulting 302 papers, we selected for detailed analysis all papers that met the following criteria: (1) it must describe a tool that supports a software engineering foraging activity, (2) the activity must have a before-navigation and an after-navigation state,

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

Table 6.6. Code set for the literature analysis. $E(V)$ = expected value, $E(C)$ = expected cost. V = actual value. C = actual cost.

and (3) the paper (or related resources) must include information of the navigation choices a programmer can make.

We then qualitatively coded the 55 papers that met these criteria based on the description of the foraging activity supported by the paper's tool (or by following references in the paper to other resources describing the tool), using the code set given in Table 6.6. As the table shows, the codes cover every possible way to align value V with $E(V)$ if $V<E(V)$, and to align cost C with $E(C)$ if $C>E(C)$ for the two factors of C, namely C_b and C_w.

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

Paper	Supports what foraging goal	Increase V	Decrease C_b	Decrease C_w	Align V & $E(V)$	Align C_b & $E(C_b)$	Align C_w & $E(C_w)$
[Alimadadi et al., 2014]	Understand interaction between source code components.	▨	▨	▨	▨		▨
[Alves et al., 2014]	Locate potential errors caused by manual refactoring.	▨	▨	▨	▨		
[Ashok et al., 2009]	Locate relevant information associated with a bug.		▨		▨		
[Baltes et al., 2014]	Locate/link sketches/diagrams relevant to part of the code.	▨	▨	▨	▨		
[Bragdon et al., 2010a]	Locate and visually organize code.	▨	▨	▨	▨		
[Caldiera & Basili, 1991]	Locate reusable code.	?	?	?		?	▨
[Coblenz et al., 2006]	Collect relevant code in a separate, easy-to-navigate patch.		▨		▨		
[Cottrell et al., 2008]	Locate reusable code.	▨	▨	▨	▨		
[Cubranic et al., 2005]	Locate artifacts relevant to the current context.	▨	▨	▨	▨		
[de Alwis & Murphy, 2008]	Locate and collect code based on programmer's query.		▨		▨	▨	
[DeLine et al., 2005a]	Find relevant code via team members' navigation histories.	▨	▨	▨	▨		▨
[Duala-Ekoko & Robillard, 2007]	Locate and collect code clones for modification.	▨	▨	▨	▨		▨
[Ducasse et al., 1999]	Locate duplicated code.		▨		▨		
[Dudziak & Wloka, 2002]	Locate potential bad code via code smells.	▨	▨	▨	▨		
[Dunn & Knight, 1993]	Locate reusable code.	?	?	?		?	?
[Fritz & Murphy, 2010]	Locate/organize code collaborated on by several programmers.		▨		▨		
[Galenson et al., 2014]	Find code snippets that meet the requirement/specification.	▨	▨	▨	▨	▨	
[Ge & Murphy-Hill, 2014]	Locate and/or fix errors caused by automated refactoring.	▨	▨	▨	▨	▨	
[Henninger, 1994]	Locate reusable code.		▨		▨		
[Hermans & Dig, 2014]	Locate source code files that may contain the bug.	▨	▨	▨	▨		
[Hill et al., 2009]	Locate relevant program elements based on NL-queries.				▨		
[Holmes et al., 2006]	Locate examples for a particular source code element.		▨	▨	▨		▨

(continued on next page)

[Holmes & Walker, 2007]	Organize and annotate code fragments during reuse tasks.		▨		▨		
[Kaleeswaran et al., 2014]	Identify changes needed to a fix a buggy piece of code.		▨	▨	▨		▨
[Kersten & Murphy, 2006]	Collect relevant code in a separate, easy-to-navigate patch.		▨	▨	▨		
[Ko, 2008]	Locate code that caused a particular output.	▨	▨	▨	▨		
[Lanubile & Visaggio, 1993]	Reverse-engineer components.		▨		?		
[Layman, 2008]	Identifies code relationships for a given code element.	▨	▨		▨		
[Lin et al., 2014]	Locate clones and identify similarities and differences.	▨	▨	▨	▨		▨
[Manotas et al., 2014]	Identify possible changes for more energy-efficient code.			▨	▨		▨
[McMillan et al., 2012]	Locate relevant software projects based on NL-query.		▨		▨		
[Mens et al., 2003]	Locate potential bad code via code smells.	▨	▨	▨	▨		
[Minto & Murphy, 2007]	Locate an expert for a given section of code.		▨	▨	▨		▨
[Mirakhorli et al., 2014]	Find code that matches a certain architectural pattern.	▨	▨	▨	▨		
[Mockus & Herbsleb, 2002]	Locate an expert for a given section of code.		▨		▨		
[Ocariza et al., 2014]	Identify possible changes to fix a buggy piece of code.		▨	▨	▨		
[Okur et al., 2014]	Identify fixes to bugs with asynch. programming constructs.	▨	▨	▨	▨		
[Olivero et al., 2011]	Locate and visually organize code.	▨	▨		▨		
[Parnin & Gorg, 2006]	Locate and recommend context-relevant code.				▨		
[Reiss, 2009]	Locate code based on programmer-supplied specification.		▨		▨		
[Schiller et al., 2014]	Identify formal behavioral specifications & document them.	▨	▨	▨	▨	▨	
[Simon et al., 2001]	Locate code suitable for refactoring via code smells.				▨		
[Storey et al., 2007]	Locate code tagged with user-determined categories.	▨	▨		▨		
[Subramanian et al., 2014]	Understand what code does via documentation & examples.	▨	▨		▨		
[Thung et al., 2014]	Locate the files that may potentially contain the bug.				▨		
[Tokuda & Batory, 2001]	Locate code for refactoring.	▨	▨	▨			

(continued on next page)

[Toomim et al., 2004]	Locate code duplicates.	▨	▨	▨			▨
[van Emden & Moonen, 2002]	Locate bad-smelling code.				▨		
[Würsch et al., 2010]	Locate code based on NL-queries.		▨		▨		
[Xiao et al., 2014]	Find relationships between classes.				▨		
[Ye et al., 2000]	Locate code for reuse.	▨	▨	▨	▨		
[Ye et al., 2007]	Locate an expert for a given section of code.	▨	▨	▨	▨		
[Zhang & Ernst, 2014]	Locate/fix configuration-related errors in a newer version.				▨		
[Zhang et al., 2014]	Identify code where similar systematic changes occurred.		▨	▨	▨		
[Zimmermann et al., 2005]	Recommend code needing modification.		▨		▨		

Table 6.7. Results of analyzing 302 papers, showing the 55 tools that assist programmers with tasks involving foraging. Shaded = some support, blank=none, ? = unclear in the paper.

6.4.2 Results

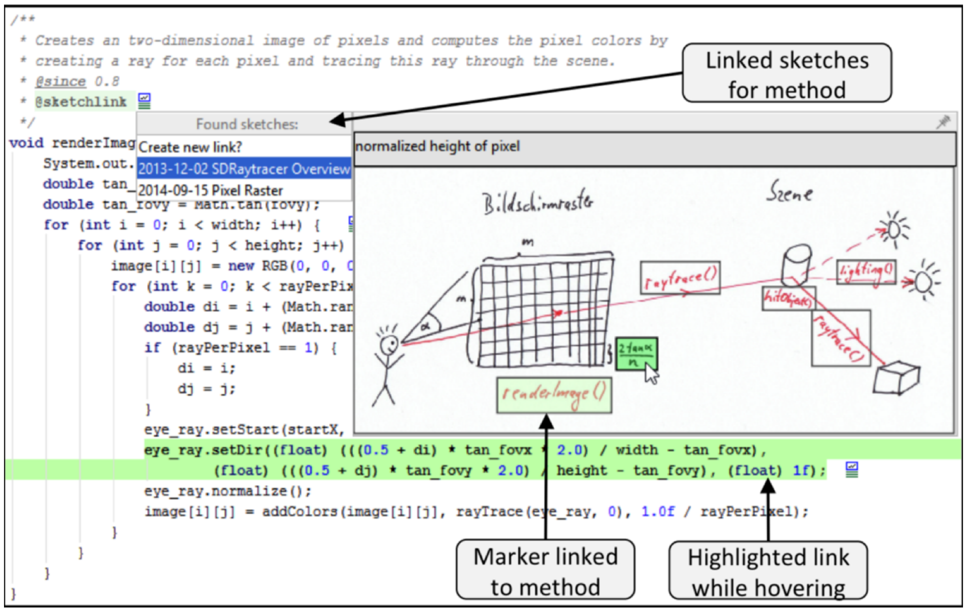
Table 6.7 presents the results of our analysis of the 55 SE research tools. As per the underlying code set (Table 6.6), Table 6.7 has a column for every possible way a tool could improve programmers' mismatches in actual versus expected value or cost, and shadings show which tools contributed to each.

A visual scan of the shaded cells in Table 6.7's columns 3-8 reveals four results. The first is good news regarding SE research's commitment to enhancing the value and cost of programmers' information seeking—100% of the 55 tools make some kind of contribution to helping programmers with aspects of value or cost.

Improving the Actuals: V and C

The second result, shown by Table 6.7's columns 3-5, is that most of these tools (47/55=85%) are working toward improving programmers' *actual* value V or cost C.

More specifically, Table 6.7's column 3 ("Increase V") shows that just over half of these 47 papers (26) work toward increasing the value V a patch delivers to programmers who make their way there. These tools do so by *adding information features* to that patch.



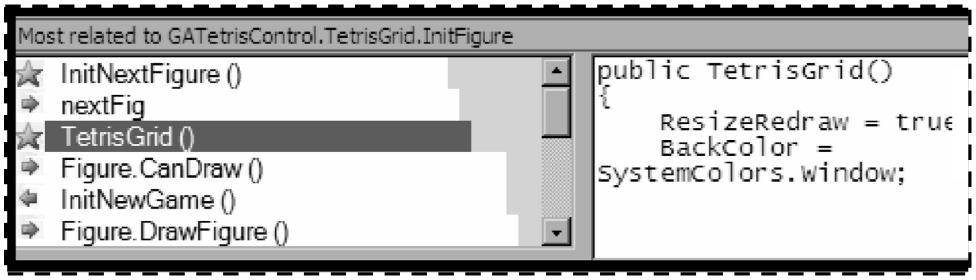
```
/**
 * Creates an two-dimensional image of pixels and computes the pixel colors by
 * creating a ray for each pixel and tracing this ray through the scene.
 * @since 0.8
 * @sketchlink
 */
void renderImag
    System.out.
    double tan_
    double tan_fovy = Math.tan(fovy);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++)
            image[i][j] = new RGB(0, 0, 0
            for (int k = 0; k < rayPerPix
                double di = i + (Math.ran
                double dj = j + (Math.ran
                if (rayPerPixel == 1) {
                    di = i;
                    dj = j;
                }
                eye_ray.setStart(startX,
                eye_ray.setDir((float) (((0.5 + di) * tan_fovx * 2.0) / width - tan_fovx),
                        (float) (((0.5 + dj) * tan_fovy * 2.0) / height - tan_fovy), (float) 1f);
                eye_ray.normalize();
                image[i][j] = addColors(image[i][j], rayTrace(eye_ray, 0), 1.0f / rayPerPixel);
            }
        }
    }
}
```

Figure 6.5. SketchLink improves V, C_b, & C_w in overlapping ways:
Increasing V: It adds information (the floating sketch) about the current method.
Decreasing C_b: Programmers can get the sketch without a tedious sequence of navigations.
Decreasing C_w: Programmer do not need to study code to infer information the sketch makes explicit.

One example is SketchLink [Baltes et al., 2014], which adds sketch diagrams relevant to the current method (Figure 6.5)—which increases V provided that these added information features help to answer the question(s) the programmer actually had, as per Research Challenge #3 (Answering the wrong question). When that provision is met, a best case is that value V to a programmer might increase from necessary up to sufficient. This best-case increase could help with Table 6.2's result that programmers' navigations did not usually produce value that was sufficient.

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

For example, SketchLink [Baltes et al., 2014] (Figure 6.5) reduces C_w by *making explicit* information the programmer would otherwise need to infer by studying the code. It also reduces C_b by *adding links* between two disjoint topologies, sketches and code, as per the two-topology case of Research Challenge #6 (Disjoint topologies).



```
public TetrisGrid()
{
    ResizeRedraw = true
    BackColor =
SystemColors.Window;
```

Figure 6.6. Team Tracks' support for both *E(V)* and *E(C_w)*: (Left) Aligning *E(V)* with *V*: Shows how often fellow team members visited a location in code as an estimate of value *V*. (Right): Aligning *E(C_w)* with C_w: If a programmer selects a method, it shows a preview, helping programmers predict how long they will spend understanding the method.

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

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

The third result is about this alignment. At first, Table 6.7 gives an impression visually that aligning *E(V)* with *V* is very common among these tools, with 52/55 (95%) making some effort to do so. However, this impression is a bit misleading, because most approaches help programmers predict patch values *only for patches that are nearby* (one navigation away), a point we shall return to shortly. Still, one excellent example of support of *E(V)* is Team Tracks [DeLine et al., 2005b], which helps programmers predict value by rating patches according to how often the programmer's team visited them (Figure 6.6, left), regardless of how many navigations away the patch is.

The fourth result Table 6.7 reveals is that tools helping to align *E(C)* with *C* were relatively rare—only 4/55 (7%) made any attempt to align *E(C_b)* with C_b, and only 10/55 (18%) worked to align *E(C_w)* with C_w. As an example of supporting *E(C_w)*, when programmers using Team Tracks select an item in the list Team Tracks recommends, it shows a preview (Figure 6.6, right), to help programmers predict the cost of understanding the code. However, as with the *E(V)* work, few tools handle patches that reside more than one navigation away.

6.4.3 The Scaling Up Problem

Section 6.4.2's examples provide useful ideas toward ultimately addressing some of the research challenges of Section 6.3, but only a few help the programmer align $E(V)$ with V or $E(C)$ with C for patches more than one click away. This leaves the programmer in a state of acute myopia (near-sightedness), unable to see beyond one navigation away—and thus unsupported in coping with long-distance problems such as those illustrated by Figure 6.2 and Figure 6.3.

This suggests our third and final open problem space:

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

Fortunately, our literature analysis points to a few notable starts in this direction. Besides the examples above, other examples are MCIDiff [Lin et al., 2014] and CloneTracker [Duala-Ekoko & Robillard, 2012]. These two clone-tracking tools consider the possible set of clones, show the length and the number of instances of *all* code clones, not just nearby ones, to help the programmer predict $E(C_w)$ of handing any or all of these clones. Another useful example is the query system Ferret [de Alwis & Murphy, 2008], which helps programmers predict $E(C_w)$ beyond the one-click-away distance. Ferret allows programmers to ask conceptual queries about a particular program element such as "What methods instantiate this type?" and while displaying the results, also shows the number of results for the query, thus allowing the programmer to gain some idea of the sum of all the $E(C_w)$'s they will incur. Promising starts like these works can serve as the ground floor upon which SE researchers can build toward ultimately addressing this problem.

6.5 Summary of Results

In this chapter, we used an Information Foraging Theory perspective to investigate programmers' navigation decisions, how often these decisions led to disappointment, and the fundamentals of why.

Open Research Problems and Challenges
Research Challenge #1: False advertising
Research Challenge #2: Synonyms
Research Challenge #3: Answering “wrong” question
Research Challenge #4: Prey in pieces
Research Challenge #5: Endless paths
Research Challenge #6: Disjoint topologies
The Scaling Up Problem

Table 6.8. Summary of open research problems and challenges.

The results suggest that how well a programmer can *predict* the value and/or cost of a navigation path are critical factors of the lower bound of a programmer’s navigation efficiency in a given information space. The results further suggest a new area of inquiry for SE researchers: how large, feature-dense SE environments can support programmers’ ability to predict the value they will receive from a navigation path and the cost they must expend to receive that value. As Sections 6.3 to 6.4.3 showed and Table 6.8 summarizes, this area of inquiry appears to be rich, challenging, and cross-cutting, with three open problem spaces involving (at least) six research challenges.

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

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

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

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

Chapter 7 Discussion and Open Questions

7.1 Between-Patch Foraging Is Difficult

Several of the findings presented in this thesis echo the same conclusion: that programmers face and endure significant challenges when foraging between patches. Our evidence shows that programmers deal with cues that often fail to fulfil their foraging needs or topologies that do not provide links to the information that they are seeking. They face unexpected costs when arriving at promising patches and sometimes give up on their foraging before they arrive at the patch or patches that contain the information they were looking for (Chapter 6).

However, despite these difficulties, and due to their specific information needs, programmers endure these challenges. The long tail from Chapter 4 showed that programmers' information needs can be very specific. That specificity manifested in P3's debugging session with a repeat patterns that repeatedly pursued the same goal. Difficulties such as these may also explain why between-patch foraging was the least common type of foraging that we observed in Chapter 4.

But why is between-patch foraging more difficult than within-patch foraging or enrichment? Within-patch foraging tends to be higher value and lower cost than between patch-foraging. In the within-patch foraging case, all the available information features are found within a single location (a method) and there are several affordances provided by the Eclipse IDE to better understand the content of the patch in a low-cost way (such as locating a variable's declaration or investigating an object's type hierarchy).

In the case of enrichment, the Eclipse IDE lowers the cost by providing several affordances to generate patches automatically, sparing the programmer the cost of finding relevant links themselves. Contrast this with between-patch foraging, where programmers often cannot assess the value and cost until they navigate to the destination patch (due to the problems discussed in Chapter 6).

Addressing the challenges of between-patch foraging likely requires addressing a combination of problems discussed in Chapter 6. To illustrate this point, consider the recommendation system from Chapter 3. Although the recommender correctly identified cues of interest to Participant 6 (thereby aligning *E(V)* with *V*), it was unable to understand the questions that the programmer was actually pursuing (effectively providing links that answered the wrong question) and did not contain topologies where the answers to those questions were (the disjoint topologies problem). For example, because the menu was not defined in the code, but in an XML properties file, the recommender was unable to provide links to the menu's definitions. Simply put, because the recommender did not consider all the relevant topologies, no amount of aligning value or cost would have led to the answer to the participant's actual question. The takeaway here is that tools designed to support between-patch navigation must go beyond "just" programmers' assessments of value and cost with the actual value and cost, and it remains a rich and interesting question.

7.2 One Missing Piece: Time

IFT's central proposition is framed in terms of the singular decision that a programmer faces at a particular moment (Chapter 2): should the programmer 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 considers the programmers *previous foraging* actions.

One body of evidence suggesting the apparent importance of the predator's previous actions. For example, in Chapter 3, we observed that navigation history (foraging momentum in the chapter) was an important predictor of where a programmer went to next. In Chapter 4, we found that participants' foraging was reactive, and the fulfillment (or abandonment) of a goal influenced the next navigation. One way this manifested was the distinct patterns of goal types, such as Repeat. In Chapter 5, we found that Learn participants were more likely to succeed by switching among different cue types at each decision point. Such results demonstrate well that programmers often do not forage in the moment; they are influenced by what they have seen before.

Another body of evidence pointing to the same thing is how a predator's previously visited patches influenced the foraging decisions the predator made in the current patch. The findings in Chapter 5 suggest that when programmers are processing cues to build information scent, they consider not only cues in the current patch (as is stipulated by IFT), but also cues that they have seen before in previously visited patches (cue types). These cases suggest the need for IFT to explicitly consider how the programmers' foraging changes over time.

7.3 Connecting the Dots: IFT and Software Engineering Research

Not only have we connected the dots between the factors and IFT's constructs, prior work has also demonstrated how SE research papers connect with IFT, thus providing additional evidence for IFT's applicability as an abstraction for software engineering research. Recall that in Section 6.4, we conducted a literature analysis that analyzed how tools to support navigation addressed value and cost across six dimensions. This coding resulted in 55 papers coded in terms of value and cost. Of these, 37 papers were sourced from [Fleming et al., 2013], where we applied IFT to identify recurring design patterns from existing software engineering research tools. However, 8 more papers from that work were not included in Section 6.4, but were explained by IFT design patterns. Thus, [Fleming et al., 2013] demonstrated how IFT abstracts across 45 papers across three types of software engineering tasks: debugging, refactoring and reuse.

Nabi et al. continued the work of [Fleming et al., 2013] and built an IFT design pattern catalog[15] authored by members of the SE research community [Nabi et al., 2016]. This catalog wiki introduced new IFT design patterns and demonstrated how an additional 66 papers (at the time of this writing) are explained by IFT's constructs and propositions. Together with the 5 papers not covered by these surveys but previously described in this thesis, a total of 128 software engineering papers so far have been connected to IFT. Table 7.1summarizes the totals.

[15] http://research.engr.oregonstate.edu/ift/readonly.php

Source	Paper citations	#
Table 6.7	[Alimadadi et al., 2014; Alves et al., 2014; Ashok et al., 2009; Baltes et al., 2014; Bragdon et al., 2010a; Caldiera & Basili, 1991; Coblenz et al., 2006; Cottrell et al., 2008; Cubranic et al., 2005; de Alwis & Murphy, 2008; DeLine et al., 2005a; Duala-Ekoko & Robillard, 2007; Ducasse et al., 1999; Dudziak & Wloka, 2002; Dunn & Knight, 1993; Fritz & Murphy, 2010; Galenson et al., 2014; Ge & Murphy-Hill, 2014; Henninger, 1994; Hermans & Dig, 2014; Hill et al., 2009; Holmes et al., 2006; Holmes & Walker, 2007; Kaleeswaran et al., 2014; Kersten & Murphy, 2006; Ko, 2008; Lanubile & Visaggio, 1993; Layman, 2008; Lin et al., 2014; Manotas et al., 2014; McMillan et al., 2012; Mens et al., 2003; Minto & Murphy, 2007; Mirakhorli et al., 2014; Mockus & Herbsleb, 2002; Ocariza et al., 2014; Okur et al., 2014; Olivero et al., 2011; Parnin & Gorg, 2006; Reiss, 2009; Schiller et al., 2014; Simon et al., 2001; Storey et al., 2007; Subramanian et al., 2014; Thung et al., 2014; Tokuda & Batory, 2001; Toomim et al., 2004; van Emden & Moonen, 2002; Würsch et al., 2010; Xiao et al., 2014; Ye et al., 2000, 2007; Zhang & Ernst, 2014; Zhang et al., 2014; Zimmermann et al., 2005]	55
Table 7.1	[Carroll, 1998; Carroll & Rosson, 1987; Grigoreanu et al., 2010; LaToza & Myers, 2010; Sillito et al., 2006]	5
Additional papers from [Fleming et al., 2013]	[Bellon et al., 2007; Dagenais & Robillard, 2010; Lanubile et al., 2010; Murphy-Hill et al., 2012; Murphy-Hill & Black, 2008; Treude & Storey, 2011, 2012; Xing & Stroulia, 2006]	8
IFT design patterns catalog	[Abreu et al., 2007; Agrawal et al., 1995; Arnold et al., 2007; Asaduzzaman et al., 2014; Beller et al., 2015; Bergel & Peña, 2014; Binkley et al., 2007; Brun et al., 2011; Campos et al., 2012; Cornelissen et al., 2008; de Oliveira Arantes & de Almeida Falbo, 2010; Duan & Cleland-Huang, 2007; Egyed et al., 2007; Eisenberg & De Volder, 2005; Gabel et al., 2008; Gall et al., 2009; Gouveia et al., 2013; Guzzi et al., 2011; Han et al., 2009; Henley & Fleming, 2014; Henninger, 1991; Hill et al., 2007; Holten, 2006; Hong et al., 2008; Hou & Pletcher, 2011; Hu & Liu, 2004; Jones & Harrold, 2005; Ko & Myers, 2009; Koenemann & Belkin, 1996; Landi, 1992; Lanza & Ducasse, 2003; LaToza & Myers, 2011; Mahmoud & Niu, 2011; Mantyla et al., 2003; McBurney & McMillan, 2016; Mockus et al., 2009; Myers & Storey, 2010; Nguyen et al., 2012a; Niu et al., 2013; Omar et al., 2012; Opdyke, 1992; Panichella et al., 2012; Pastore et al., 2013; Perez & Abreu, 2013, 2014; Qi et al., 2012; Ren et al., 2004; Safyallah & Sartipi, 2006; Savage et al., 2010; Scaffidi, 2010; Servant & Jones, 2012; Stolee et al., 2014; Thummalapenta et al., 2011; Treude et al., 2011; Treude & Storey, 2010; Weiser, 1981; Wilde & Scully, 1995; Yu et al., 2012; Zhang & Hou, 2013; Zhao et al., 2004]	60
Total		128

Table 7.1. 128 SE research papers that are connected using IFT's abstractions.

7.4 Generalizability

Any work that builds theory needs to show that it generalizes beyond the initial situation in which it was developed.

Thus, here we summarize aspects of this work for which we have evaluated generalizability. In Chapter 3, we showed that predictive factors that previously were evaluated in offline situations (i.e., in predictive models) generalized to an in-context situation with different participants (the recommender tool). The foraging goals in Chapter 4 started with the goals that Sillito's work had already reported from a wide spectrum of environments and situations on questions that programmers ask [Sillito et al., 2006] and reapplied them to IFT's notions of diet and information goals in a new empirical situation. Chapter 5's extension of cues into cue types has recently been generalized into Srinivasa Ragavan et al.'s [Srinivasa Ragavan et al., 2016] work on information foraging with variants. In that work, she reused cue types to explain how programmers foraged across variants of a JavaScript program (not Java programs as in our work) in a new IDE (Cloud9, not the Eclipse environment in our work). Finally, Chapter 5 showed that Fixers versus Learners forage significantly differently in two different environments (Eclipse and AIDE) on two different platforms (Desktop versus Mobile) given two different programs (jEdit vs. Vanilla Music) in different domains (a text editor versus a music player).

7.5 Threats to Validity

As in any empirical research, the results in this thesis may have been influenced by the environments the participants used, the tools available to them, the tasks they worked on, etc. However, our methodology was designed to strengthen generalizability through the use of code sets and methodological conventions from other pertinent studies (e.g., [Grigoreanu et al., 2010, 2012; Romero et al., 2007; Sillito et al., 2006]), and through the use of realistic elements: both jEdit and Vanilla Music are real open-source projects; the defects were sourced for actual bug repositories; the participants were experienced professionals (except Desktop participants in Chapter 5) using a popular IDE (Eclipse or AIDE).

The primary threat to external validity of the empirical studies is that all studies' participants were new to the code base. This is a common situation for new hires and when programmers transfer to different development teams, but we do not expect results to generalize to other kinds of debugging situations. All our studies required the participants to use the Java programming language, thus there is a question of generalizability to other programming. Finally, although IDEs share many of the same features, our findings may not generalize beyond Eclipse and AIDE.

We guarded against threats to internal validity in several ways. For our participant studies (Chapter 3, Chapter 4, Chapter 5 and Chapter 6), our Jaccard inter-rater reliability was at least 80% on all code sets. To help assure construct validity (the extent to which a measure actually captures what is intended), we did not rely on our raters' interpretations of participants alone, but also used follow-up questionnaires and retrospective interviews. In the retrospectives, played back videos of the task sessions to remind participants where they were and also to gather participants' interpretations of which events mattered and why they did what they did (Chapter 4, Chapter 5, and Chapter 6). For Chapter 3, we instead triangulated using a follow-up questionnaire. However, participants' recollections of what happened later may have biased their responses. Issues like these can be resolved only through additional studies.

Chapter 8 Conclusion

Without theory, ad-hoc tool development aiming to support programmer navigation limits our ability to explain and to leverage promising ways to assist programmers' foraging. Information Foraging Theory provides a solution. IFT provides a theoretical foundation to unify those tools' various approaches under a single abstraction, i.e., to "connect the dots" – enabling understanding of why a particular approach does or does not work. Therefore, in this dissertation, we showed how IFT can help unify SE research spanning multiple software engineering disciplines.

First, we evaluated predictive factors of programmer navigation in situ by building and empirically investigating a recommendation tool to support programmer navigations. We found that factors underlying the tools' models fundamentally served as different types of scent. This led to insights into what types of scent matter to programmers' foraging. Our findings suggest further that as new tools are developed, new types of scent will emerge, thus further improving our understanding of how programmers estimate value and cost while foraging.

Second, we investigated programmers' foraging goals and strategies and the relationship between them from the perspective of IFT's diet construct. We found that foraging goal types tended to be pursued more often by certain foraging strategies, yet IFT makes no mention of this relationship. Without making relationships like this explicit, consumers of IFT may find it challenging to find the best way to leverage the theory.

Third, we investigated the role of Minimal Learning Theory's production bias on programmers' foraging behavior finding that it influenced programmers' scent, affecting which patch types they navigated from and the cue types that they attended to. This was the first work to consider IFT's constructs of patches and cues that was not dependent on their content, potentially simplifying future models for IFT-based tools. Additionally, in terms of IFT's utility, we showed that parts of IFT findings for debugging generalized beyond jEdit and beyond Eclipse.

Fourth, we found that programmers faced significant difficulties estimating scent, resulting in about 51% of participants' navigations ending in some degree of disappointment in the information value they received and higher than expected costs in about 37% of their navigations. Digging deeper, we identified seven IFT-based open research problems revolving around value and costs in foraging.

Together, these findings highlight how to connect the dots between the ways that software engineering research tools and studies explain programmers' foraging and what those works' findings mean for IFT. We view this work as step towards advancing IFT as a foundation to abstract programmers' information seeking under a unifying theory. Hopefully, these insights provide useful information for those looking to leverage IFT to better support programmers' foraging in the future.

References

Abreu, R., Zoeteweij, P., & van Gemund, A. J. C. (2007). On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007* (pp. 89–98). http://doi.org/10.1109/TAIC.PART.2007.13

Agrawal, H., Horgan, J. R., London, S., & Wong, W. E. (1995). Fault localization using execution slices and dataflow tests. In *, Sixth International Symposium on Software Reliability Engineering, 1995. Proceedings* (pp. 143–151). http://doi.org/10.1109/ISSRE.1995.497652

Agresti, A., & Kateri, M. (2011). *Categorical data analysis*. Springer. Retrieved from http://link.springer.com/10.1007/978-3-642-04898-2_161

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers Principles, Techniques, and Tools*. Addison Wesley. Retrieved from http://140.118.105.174/Courses/SE/2012/14_WebService.pdf

Alimadadi, S., Sequeira, S., Mesbah, A., & Pattabiraman, K. (2014). Understanding JavaScript Event-based Interactions. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 367–377). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568268

Alves, E. L. G., Song, M., & Kim, M. (2014). RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 751–754). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661674

Anderson, J. (1990). *The adaptive character of thought*. Psychology Press. Retrieved from https://books.google.com/books?hl=en&lr=&id=Vp_wcfyIKH0C&oi=fnd&pg=PR3&dq=The+Adaptive+Character+of+Thought&ots=1OCagYlukX&sig=KxthYXPfwgVRknbmAYuuWQftrms

Anderson, J. (1993). *Rules of the mind*. Psychology Press. Retrieved from https://books.google.com/books?hl=en&lr=&id=1KOYAgAAQBAJ&oi=fnd&pg=PP1&dq=Rules+of+the+Mind&ots=jIzD1FTwdY&sig=sH46Dd8z-8SwAAdoM0_sEt0A2hk

Anderson, J., & Pirolli, P. L. (1984). Spread of activation. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*(4), 791.

Arnold, D. C., Ahn, D. H., de Supinski, B. R., Lee, G. L., Miller, B. P., & Schulz, M. (2007). Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium* (pp. 1–10). http://doi.org/10.1109/IPDPS.2007.370254

Asaduzzaman, M., Roy, C. K., Schneider, K. A., & Hou, D. (2014). CSCC: Simple, Efficient, Context Sensitive Code Completion. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 71–80). http://doi.org/10.1109/ICSME.2014.29

Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., & Vangala, V. (2009). DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (pp. 373–382). New York, NY, USA: ACM. http://doi.org/10.1145/1595696.1595766

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval: The Concepts and Technology Behind Search*. ACM. Retrieved from ftp://mail.im.tku.edu.tw/seke/slide/baeza-yates/chap10_user_interfaces_and_visualization-modern_ir.pdf

Bajracharya, S. K., & Lopes, C. V. (2012). Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, *17*(4-5), 424–466. http://doi.org/10.1007/s10664-010-9144-6

Baltes, S., Schmitz, P., & Diehl, S. (2014). Linking Sketches and Diagrams to Source Code Artifacts. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 743–746). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661672

Beller, M., Gousios, G., & Zaidman, A. (2015). How (Much) Do Developers Test? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 2, pp. 559–562). http://doi.org/10.1109/ICSE.2015.193

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, *33*(9), 577–591. http://doi.org/10.1109/TSE.2007.70725

Bergel, A., & Peña, V. (2014). Increasing test coverage with Hapao. *Science of Computer Programming*, *79*, 86–100. http://doi.org/10.1016/j.scico.2012.04.006

Binkley, D., Gold, N., & Harman, M. (2007). An Empirical Study of Static Program Slice Size. *ACM Trans. Softw. Eng. Methodol.*, *16*(2). http://doi.org/10.1145/1217295.1217297

Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., … LaViola, J., Jr. (2010a). Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (pp. 455–464). New York, NY, USA: ACM. http://doi.org/10.1145/1806799.1806866

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., … LaViola, J., Jr. (2010b). Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human*

Factors in Computing Systems (pp. 2503–2512). New York, NY, USA: ACM. http://doi.org/10.1145/1753326.1753706

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1589–1598). New York, NY, USA: ACM. http://doi.org/10.1145/1518701.1518944

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. http://doi.org/10.1016/S0020-7373(83)80031-5

Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2011). Crystal: Precise and Unobtrusive Conflict Warnings. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 444–447). New York, NY, USA: ACM. http://doi.org/10.1145/2025113.2025187

Caldiera, G., & Basili, V. R. (1991). Identifying and qualifying reusable software components. *Computer*, *24*(2), 61–70. http://doi.org/10.1109/2.67210

Campos, J., Riboira, A., Perez, A., & Abreu, R. (2012). GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 378–381). New York, NY, USA: ACM. http://doi.org/10.1145/2351676.2351752

Card, S. K., Pirolli, P. L., Van Der Wege, M., Morrison, J., Reeder, R., Schraedley, P., & Boshart, J. (2001). Information Scent As a Driver of Web Behavior Graphs: Results of a Protocol Analysis Method for Web Usability. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 498–505). New York, NY, USA: ACM. http://doi.org/10.1145/365024.365331

Carroll, J. (1998). *Minimalism beyond the Nurnberg funnel*. MIT Press. Retrieved from https://books.google.com/books?hl=en&lr=&id=LvXiZJEUJjAC&oi=fnd&pg=PR11&dq=Minimalism+Beyond+the+Nurnberg+Funnel,&ots=1lnD2P1lyk&sig=tOw4nV8DVQPKebRUYZrQK7e7F0A

Carroll, J., & Rosson, M. B. (1987). *Paradox of the Active User*. The MIT Press. Retrieved from http://psycnet.apa.org/psycinfo/1987-98055-005

Chen, X., & Grundy, J. (2011). Improving Automated Documentation to Code Traceability by Combining Retrieval Techniques. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 223–232). Washington, DC, USA: IEEE. http://doi.org/10.1109/ASE.2011.6100057

Chi, E. H., & Pirolli, P. L. (2006). Social information foraging and collaborative search. In *Human Computer Interaction Consortium*. Fraser, Colorado, USA. Retrieved from https://www.researchgate.net/profile/Peter_Pirolli/publication/247563469_Social_Infor-mation_Foraging_and_Collaborative_Search/links/02e7e52965b7e05894000000.pdf

Chi, E. H., Pirolli, P. L., Chen, K., & Pitkow, J. (2001). Using Information Scent to Model User Information Needs and Actions and the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 490–497). New York, NY, USA: ACM. http://doi.org/10.1145/365024.365325

Chi, E. H., Pirolli, P. L., & Pitkow, J. (2000). The Scent of a Site: A System for Analyzing and Predicting Information Scent, Usage, and Usability of a Web Site. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 161–168). New York, NY, USA: ACM. http://doi.org/10.1145/332040.332423

Chi, E. H., Rosien, A., Supattanasiri, G., Williams, A., Royer, C., Chow, C., … Cousins, S. (2003). The Bloodhound Project: Automating Discovery of Web Usability Issues Using the InfoScent Simulator. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 505–512). New York, NY, USA: ACM. http://doi.org/10.1145/642611.642699

Coblenz, M. J., Ko, A. J., & Myers, B. A. (2006). JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange* (pp. 65–69). New York, NY, USA: ACM. http://doi.org/10.1145/1188835.1188849

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., & van Wijk, J. J. (2008). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, *81*(12), 2252–2268. http://doi.org/10.1016/j.jss.2008.02.068

Cottrell, R., Walker, R. J., & Denzinger, J. (2008). Semi-automating Small-scale Source Code Reuse via Structural Correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 214–225). New York, NY, USA: ACM. http://doi.org/10.1145/1453101.1453130

Cubranic, D., & Murphy, G. C. (2003). Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 408–418). ACM/IEEE. http://doi.org/10.1109/ICSE.2003.1201219

Cubranic, D., Murphy, G. C., Singer, J., & Booth, K. S. (2005). Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering*, *31*(6), 446–465. http://doi.org/10.1109/TSE.2005.71

Dagenais, B., & Robillard, M. P. (2010). Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of*

Software Engineering (pp. 127–136). New York, NY, USA: ACM. http://doi.org/10.1145/1882291.1882312

de Alwis, B., & Murphy, G. C. (2008). Answering conceptual queries with Ferret. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 21–30). ACM/IEEE. http://doi.org/10.1145/1368088.1368092

DeLine, R., Czerwinski, M., & Robertson, G. (2005a). Easing program comprehension by sharing navigation data. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)* (pp. 241–248). http://doi.org/10.1109/VLHCC.2005.32

DeLine, R., Khella, A., Czerwinski, M., & Robertson, G. (2005b). Towards Understanding Programs Through Wear-based Filtering. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (pp. 183–192). New York, NY, USA: ACM. http://doi.org/10.1145/1056018.1056044

de Oliveira Arantes, L., & de Almeida Falbo, R. (2010). An Infrastructure for Managing Semantic Documents. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops* (pp. 235–244). http://doi.org/10.1109/EDOCW.2010.17

Duala-Ekoko, E., & Robillard, M. P. (2007). Tracking Code Clones in Evolving Software. In *29th International Conference on Software Engineering (ICSE'07)* (pp. 158–167). http://doi.org/10.1109/ICSE.2007.90

Duala-Ekoko, E., & Robillard, M. P. (2012). Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 266–276). Piscataway, NJ, USA: ACM/IEEE. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337255

Duan, C., & Cleland-Huang, J. (2007). Clustering Support for Automated Tracing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (pp. 244–253). New York, NY, USA: ACM. http://doi.org/10.1145/1321631.1321668

Ducasse, S., Rieger, M., & Demeyer, S. (1999). A language independent approach for detecting duplicated code. In *IEEE International Conference on Software Maintenance, 1999. (ICSM '99) Proceedings* (pp. 109–118). http://doi.org/10.1109/ICSM.1999.792593

Dudziak, T., & Wloka, J. (2002). *Tool-supported discovery and refactoring of structural weaknesses in code*. Retrieved from http://wloka.org/publications_files/dudwlo02-jart-master-thesis.pdf

Dunn, M. F., & Knight, J. C. (1993). Automating the Detection of Reusable Parts in Existing Software. In *Proceedings of the 15th International Conference on Software Engineering* (pp. 381–390). Los Alamitos, CA, USA: IEEE Computer Society Press. Retrieved from http://dl.acm.org/citation.cfm?id=257572.257660

Egyed, A., Binder, G., & Grunbacher, P. (2007). STRADA: A Tool for Scenario-Based Feature-to-Code Trace Detection and Analysis. In *Companion to the Proceedings of the 29th International Conference on Software Engineering* (pp. 41–42). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICSECOMPANION.2007.70

Eisenberg, A. D., & De Volder, K. (2005). Dynamic feature traces: finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 337–346). http://doi.org/10.1109/ICSM.2005.42

Evans, B., & Card, S. (2008a). Augmented Information Assimilation: Social and Algorithmic Web Aids for the Information Long Tail. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 989–998). New York, NY, USA: ACM. http://doi.org/10.1145/1357054.1357207

Evans, B., & Card, S. K. (2008b). Augmented Information Assimilation: Social and Algorithmic Web Aids for the Information Long Tail. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 989–998). New York, NY, USA: ACM. http://doi.org/10.1145/1357054.1357207

Evans, B., Kairam, S., & Pirolli, P. L. (2010). Do your friends make you smarter?: An analysis of social strategies in online information seeking. *Information Processing & Management*, *46*(6), 679–692. http://doi.org/10.1016/j.ipm.2009.12.001

Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., & Kwan, I. (2013). An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering Methodologies*, *22*(2), 14:1–14:41. http://doi.org/10.1145/2430545.2430551

Fritz, T., & Murphy, G. C. (2010). Using Information Fragments to Answer the Questions Developers Ask. In *Proceedings of the 32nd International Conference on Software Engineering* (pp. 175–184). New York, NY, USA: ACM. http://doi.org/10.1145/1806799.1806828

Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1987). The Vocabulary Problem in Human-system Communication. *Commun. ACM*, *30*(11), 964–971. http://doi.org/10.1145/32206.32212

Fu, W.-T., & Pirolli, P. L. (2007). SNIF-ACT: A Cognitive Model of User Navigation on the World Wide Web. *Human–Computer Interaction*, *22*(4), 355–412. http://doi.org/10.1080/07370020701638806

Gabel, M., Jiang, L., & Su, Z. (2008). Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering* (pp. 321–330). http://doi.org/10.1145/1368088.1368132

Galenson, J., Reames, P., Bodik, R., Hartmann, B., & Sen, K. (2014). CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th Interna-*

tional Conference on Software Engineering (pp. 653–663). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568250

Gall, H. C., Fluri, B., & Pinzger, M. (2009). Change Analysis with Evolizer and ChangeDistiller. *IEEE Softw.*, *26*(1), 26–33. http://doi.org/10.1109/MS.2009.6

Ge, X., & Murphy-Hill, E. (2014). Manual Refactoring Changes with Automated Refactoring Validation. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 1095–1105). New York, NY, USA: ACM/IEEE. http://doi.org/10.1145/2568225.2568280

Gouveia, C., Campos, J., & Abreu, R. (2013). Using HTML5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 1–10). http://doi.org/10.1109/VISSOFT.2013.6650539

Grigoreanu, V., Burnett, M., & Robertson, G. (2010). A Strategy-centric Approach to the Design of End-user Debugging Tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 713–722). New York, NY, USA: ACM. http://doi.org/10.1145/1753326.1753431

Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., & Kwan, I. (2012). End-user Debugging Strategies: A Sensemaking Perspective. *ACM Trans. Comput.-Hum. Interact.*, *19*(1), 5:1–5:28. http://doi.org/10.1145/2147783.2147788

Guzzi, A., Hattori, L., Lanza, M., Pinzger, M., & Deursen, A. v. (2011). Collective Code Bookmarks for Program Comprehension. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)* (pp. 101–110). http://doi.org/10.1109/ICPC.2011.19

Han, S., Wallace, D. R., & Miller, R. C. (2009). Code Completion from Abbreviated Input. In *24th IEEE/ACM International Conference on Automated Software Engineering, 2009. ASE '09* (pp. 332–343). http://doi.org/10.1109/ASE.2009.64

Hearst, M. (2011). User interfaces for search. *Modern Information Retrieval*, 21–55.

Henley, A. Z., & Fleming, S. D. (2014). The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems* (pp. 2511–2520). New York, NY, USA: ACM. http://doi.org/10.1145/2556288.2557073

Henninger, S. (1991). CodeFinder: A Tool For Locating Software Objects For Reuse. In *Automating Software Design: Interactive Design Workshop Notes AAAI–91 pp* (pp. 40–47).

Henninger, S. (1994). Using iterative refinement to find reusable software. *IEEE Software*, *11*(5), 48–59. http://doi.org/10.1109/52.311059

Hermans, F., & Dig, D. (2014). BumbleBee: A Refactoring Environment for Spreadsheet Formulas. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium* *on Foundations of Software Engineering* (pp. 747–750). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661673

Hill, E., Pollock, L., & Vijay-Shanker, K. (2007). Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (pp. 14–23). New York, NY, USA: ACM. http://doi.org/10.1145/1321631.1321637

Hill, E., Pollock, L., & Vijay-Shanker, K. (2009). Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 232–242). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICSE.2009.5070524

Hill, W. C., Hollan, J. D., Wroblewski, D., & McCandless, T. (1992). Edit Wear and Read Wear. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3–9). New York, NY, USA: ACM. http://doi.org/10.1145/142750.142751

Holmes, R., & Walker, R. J. (2007). Supporting the Investigation and Planning of Pragmatic Reuse Tasks. In *Proceedings of the 29th International Conference on Software Engineering* (pp. 447–457). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICSE.2007.83

Holmes, R., Walker, R. J., & Murphy, G. C. (2006). Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Transactions on Software Engineering*, *32*(12), 952–970. http://doi.org/10.1109/TSE.2006.117

Holten, D. (2006). Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, *12*(5), 741–748. http://doi.org/10.1109/TVCG.2006.147

Hong, L., Chi, E. H., Budiu, R., Pirolli, P., & Nelson, L. (2008). SparTag.Us: A Low Cost Tagging System for Foraging of Web Content. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (pp. 65–72). New York, NY, USA: ACM. http://doi.org/10.1145/1385569.1385582

Hou, D., & Pletcher, D. M. (2011). An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (pp. 233–242). http://doi.org/10.1109/ICSM.2011.6080790

Hu, M., & Liu, B. (2004). Mining and Summarizing Customer Reviews. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 168–177). New York, NY, USA: ACM. http://doi.org/10.1145/1014052.1014073

Jakobsen, M. R., & Hornbæk, K. (2006). Evaluating a Fisheye View of Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 377–386). New York, NY, USA: ACM. http://doi.org/10.1145/1124772.1124830

John, B. E., Swart, C., Bellamy, R., Blackmon, M. H., & Brown, R. (2013). An Open Source Approach to Information Scent. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems* (pp. 355–360). New York, NY, USA: ACM. http://doi.org/10.1145/2468356.2468419

Jones, J. A., & Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (pp. 273–282). New York, NY, USA: ACM. http://doi.org/10.1145/1101908.1101949

Kaleeswaran, S., Tulsian, V., Kanade, A., & Orso, A. (2014). MintHint: Automated Synthesis of Repair Hints. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 266–276). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568258

Karrer, T., Krämer, J.-P., Diehl, J., Hartmann, B., & Borchers, J. (2011). Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (pp. 217–224). New York, NY, USA: ACM. http://doi.org/10.1145/2047196.2047225

Kersten, M., & Murphy, G. C. (2005). Mylar: A Degree-of-interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development* (pp. 159–168). New York, NY, USA: ACM. http://doi.org/10.1145/1052898.1052912

Kersten, M., & Murphy, G. C. (2006). Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 1–11). New York, NY, USA: ACM. http://doi.org/10.1145/1181775.1181777

Kevic, K., Fritz, T., & Shepherd, D. C. (2014). CoMoGen: An Approach to Locate Relevant Task Context by Combining Search and Navigation. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 61–70). Washington, DC, USA: IEEE. http://doi.org/10.1109/ICSME.2014.28

Kittur, A., Peters, A. M., Diriye, A., Telang, T., & Bove, M. R. (2013). Costs and Benefits of Structured Information Foraging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2989–2998). New York, NY, USA: ACM. http://doi.org/10.1145/2470654.2481415

Ko, A. J. (2008). *Asking and Answering Questions About the Causes of Software Behavior*. Carnegie Mellon University, Pittsburgh, PA, USA.

Ko, A. J., & Myers, B. A. (2008). Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 301–310). New York, NY, USA: ACM/IEEE. http://doi.org/10.1145/1368088.1368130

Ko, A. J., & Myers, B. A. (2009). Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1569–1578). New York, NY, USA: ACM. http://doi.org/10.1145/1518701.1518942

Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, *32*(12), 971–987. http://doi.org/10.1109/TSE.2006.116

Koenemann, J., & Belkin, N. J. (1996). A Case for Interaction: A Study of Interactive Information Retrieval Behavior and Effectiveness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 205–212). New York, NY, USA: ACM. http://doi.org/10.1145/238386.238487

Krämer, J.-P., Karrer, T., Kurz, J., Wittenhagen, M., & Borchers, J. (2013). How Tools in IDEs Shape Developers' Navigation Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3073–3082). New York, NY, USA: ACM. http://doi.org/10.1145/2470654.2466419

Kuttal, S. K., Sarma, A., & Rothermel, G. (2013). Predator behavior in the wild web world of bugs: An information foraging theory perspective. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing* (pp. 59–66). IEEE. http://doi.org/10.1109/VLHCC.2013.6645244

Landi, W. (1992). Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, *1*(4), 323–337. http://doi.org/10.1145/161494.161501

Lanubile, F., Ebert, C., Prikladnicki, R., & Vizcaíno, A. (2010). Collaboration Tools for Global Software Engineering. *IEEE Softw.*, *27*(2), 52–55. http://doi.org/10.1109/MS.2010.39

Lanubile, F., & Visaggio, G. (1993). Function recovery based on program slicing. In *Proceedings of the 1993 Conference on Software Maintenance* (pp. 396–404). http://doi.org/10.1109/ICSM.1993.366923

Lanza, M., & Ducasse, S. (2003). Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, *29*(9), 782–795. http://doi.org/10.1109/TSE.2003.1232284

LaToza, T. D., & Myers, B. A. (2010). Developers Ask Reachability Questions. In *Proceedings of the 32nd International Conference on Software Engineering* (pp. 185–194). New York, NY, USA: ACM. http://doi.org/10.1145/1806799.1806829

LaToza, T. D., & Myers, B. A. (2011). Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 117–124). IEEE. http://doi.org/10.1109/VLHCC.2011.6070388

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering* (pp. 492–501). New York, NY, USA: ACM/IEEE. http://doi.org/10.1145/1134285.1134355

Lawrance, J., Bellamy, R., & Burnett, M. (2007). Scents in Programs:Does Information Foraging Theory Apply to Program Maintenance? In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 15–22). IEEE. http://doi.org/10.1109/VLHCC.2007.25

Lawrance, J., Bellamy, R., Burnett, M., & Rector, K. (2008a). Can information foraging pick the fix? A field study. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 57–64). http://doi.org/10.1109/VLHCC.2008.4639059

Lawrance, J., Bellamy, R., Burnett, M., & Rector, K. (2008b). Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1323–1332). New York, NY, USA: ACM. http://doi.org/10.1145/1357054.1357261

Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., & Fleming, S. D. (2013). How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*, *39*(2), 197–215. http://doi.org/10.1109/TSE.2010.111

Lawrance, J., Burnett, M., Bellamy, R., Bogart, C., & Swart, C. (2010). Reactive Information Foraging for Evolving Goals. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 25–34). New York, NY, USA: ACM. http://doi.org/10.1145/1753326.1753332

Layman, L. (2008, December 16). *Information Needs of Developers for Program Comprehension during Software Maintenance Tasks*. North Carolina State University.

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software*, *7*(4), 325–339. http://doi.org/10.1016/0164-1212(87)90032-X

Lin, Y., Xing, Z., Xue, Y., Liu, Y., Peng, X., Sun, J., & Zhao, W. (2014). Detecting Differences Across Multiple Instances of Code Clones. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 164–174). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568298

Luca, L., Stephen, B., & Pierpaolo, D. (2009). Information Foraging Theory as a Form of Collective Intelligence for Social Search. In N. T. Nguyen, R. Kowalczyk, & S.-M. Chen (Eds.), *Computational Collective Intelligence. Semantic Web, Social*

Networks and Multiagent Systems (pp. 63–74). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-04441-0_5

Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2014). On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.*, *23*(4), 31:1–31:37. http://doi.org/10.1145/2622669

Mahmoud, A., & Niu, N. (2011). TraCter: A tool for candidate traceability link clustering. In *2011 IEEE 19th International Requirements Engineering Conference* (pp. 335–336). http://doi.org/10.1109/RE.2011.6051663

Majid, I., & Robillard, M. P. (2005). NaCIN: An Eclipse Plug-in for Program Navigation-based Concern Inference. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange* (pp. 70–74). New York, NY, USA: ACM. http://doi.org/10.1145/1117696.1117711

Manotas, I., Pollock, L., & Clause, J. (2014). SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 503–514). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568297

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings* (pp. 381–384). http://doi.org/10.1109/ICSM.2003.1235447

McBurney, P. W., & McMillan, C. (2016). An empirical study of the textual similarity between source code and source code summaries. *Empirical Software Engineering*, *21*(1), 17–42. http://doi.org/10.1007/s10664-014-9344-6

McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., & Xie, Q. (2012). Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering*, *38*(5), 1069–1087. http://doi.org/10.1109/TSE.2011.84

Mens, T., Tourwé, T., & Muñoz, F. (2003). Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. In *Proceedings of the 6th International Workshop on Principles of Software Evolution* (p. 39–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=942803.943730

Minto, S., & Murphy, G. C. (2007). Recommending Emergent Teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)* (pp. 5–5). http://doi.org/10.1109/MSR.2007.27

Mirakhorli, M., Fakhry, A., Grechko, A., Wieloch, M., & Cleland-Huang, J. (2014). Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium* *on Foundations of Software Engineering* (pp. 739–742). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661671

Mockus, A., & Herbsleb, J. D. (2002). Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the 24th International Conference on Software Engineering* (pp. 503–512). New York, NY, USA: ACM. http://doi.org/10.1145/581339.581401

Mockus, A., Nagappan, N., & Dinh-Trong, T. T. (2009). Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement* (pp. 291–301). http://doi.org/10.1109/ESEM.2009.5315981

Murphy-Hill, E., & Black, A. P. (2008). Refactoring Tools: Fitness for Purpose. *IEEE Software*, *25*(5), 38–44. http://doi.org/10.1109/MS.2008.123

Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, *38*(1), 5–18. http://doi.org/10.1109/TSE.2011.41

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 163–167). New York, NY, USA: ACM. http://doi.org/10.1145/1352135.1352191

Myers, D., & Storey, M.-A. (2010). Using Dynamic Analysis to Create Trace-focused User Interfaces for IDEs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 367–368). New York, NY, USA: ACM. http://doi.org/10.1145/1882291.1882351

Nabi, T., Sweeney, K. M. D., Lichlyter, S., Piorkowski, D., Scaffidi, C., Burnett, M., & Fleming, S. D. (2016). Putting Information Foraging Theory to Work: Community-based Design Patterns for Programming Tools. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing* (p. 5). IEEE.

Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., & Nguyen, T. N. (2012a). Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 69–79). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337232

Nguyen, T. A., Rumee, S. T. A., Csallner, C., & Tillmann, N. (2012b). An Experiment in Developing Small Mobile Phone Applications Comparing On-phone to Off-phone Development. In *Proceedings of the First International Workshop on User Evaluation for Software Engineering Researchers* (pp. 9–12). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2667089.2667092

Nielsen, J. (2003, June 30). Information Foraging: Why Google Makes People Leave Your Site Faster. Retrieved July 24, 2016, from https://www.nngroup.com/articles/information-scent/

Niu, N., Mahmoud, A., & Bradshaw, G. (2011). Information Foraging As a Foundation for Code Navigation. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 816–819). New York, NY, USA: ACM/IEEE. http://doi.org/10.1145/1985793.1985911

Niu, N., Mahmoud, A., Chen, Z., & Bradshaw, G. (2013). Departures from Optimality: Understanding Human Analyst's Information Foraging in Assisted Requirements Tracing. In *Proceedings of the 35th International Conference on Software Engineering* (pp. 572–581). Piscataway, NJ, USA: ACM/IEEE. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486864

Ocariza, F. S., Jr., Pattabiraman, K., & Mesbah, A. (2014). Vejovis: Suggesting Fixes for JavaScript Faults. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 837–847). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568257

Okur, S., Hartveld, D. L., Dig, D., & Deursen, A. van. (2014). A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 1117–1127). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568309

Olivero, F., Lanza, M., D'Ambros, M., & Robbes, R. (2011). Enabling program comprehension through a visual object-focused development environment. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 127–134). http://doi.org/10.1109/VLHCC.2011.6070389

Olston, C., & Chi, E. H. (2003). ScentTrails: Integrating Browsing and Searching on the Web. *ACM Transactions on Computer-Human Interaction*, *10*(3), 177–197. http://doi.org/10.1145/937549.937550

Omar, C., Yoon, Y., LaToza, T. D., & Myers, B. A. (2012). Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 859–869). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337324

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign. Retrieved from http://embedded.cs.ccu.edu.tw/OldVersionWebPages/vertaf/internal/wares/Quantum%20Framework/Resources/Opdyke92.pdf

Panichella, S., Aponte, J., di Penta, M., Marcus, A., & Canfora, G. (2012). Mining source code descriptions from developer communications. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)* (pp. 63–72). http://doi.org/10.1109/ICPC.2012.6240510

Parnin, C., & Gorg, C. (2006). Building Usage Contexts During Program Comprehension. In *14th IEEE International Conference on Program Comprehension (ICPC'06)* (pp. 13–22). IEEE. http://doi.org/10.1109/ICPC.2006.14

Pastore, F., Mariani, L., & Goffi, A. (2013). RADAR: A tool for debugging regression problems in C/C #x002B; #x002B; Software. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 1335–1338). http://doi.org/10.1109/ICSE.2013.6606711

Perez, A., & Abreu, R. (2013). Cues for scent intensification in debugging. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 120–125). http://doi.org/10.1109/ISSREW.2013.6688890

Perez, A., & Abreu, R. (2014). A Diagnosis-based Approach to Software Comprehension. In *Proceedings of the 22Nd International Conference on Program Comprehension* (pp. 37–47). New York, NY, USA: ACM. http://doi.org/10.1145/2597008.2597151

Piorkowski, D. (2013, November 18). *Modeling programmer navigation : an empirical evaluation of predictive models*. Retrieved from http://ir.library.oregonstate.edu/xmlui/handle/1957/44671

Piorkowski, D., Fleming, S. D., Kwan, I., Burnett, M., Scaffidi, C., Bellamy, R., & Jordahl, J. (2013). The Whats and Hows of Programmers' Foraging Diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3063–3072). New York, NY, USA: ACM. http://doi.org/10.1145/2470654.2466418

Piorkowski, D., Fleming, S. D., Scaffidi, C., Bogart, C., Burnett, M., John, B. E., … Swart, C. (2012). Reactive Information Foraging: An Empirical Investigation of Theory-based Recommender Systems for Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1471–1480). New York, NY, USA: ACM. http://doi.org/10.1145/2207676.2208608

Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., … Bellamy, R. (2011). Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 109–116). IEEE. http://doi.org/10.1109/VLHCC.2011.6070387

Pirolli, P. L. (1998). Exploring Browser Design Trade-offs Using a Dynamical Model of Optimal Information Foraging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 33–40). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. http://doi.org/10.1145/274644.274650

Pirolli, P. L. (2005). Rational analyses of information foraging on the web. *Cognitive Science*, *29*(3), 343–373.

Pirolli, P. L. (2007). *Information foraging theory: Adaptive interaction with information*. Oxford University Press. Retrieved from https://books.google.com/books?hl=en&lr=&id=LADEE_1fwLQC&oi=fnd&pg=PR11&dq=information+foraging+theory&ots=NpZR7wwxF_&sig=3ydAT-wm0c3VNvfklXDiL9RKhlE

Pirolli, P. L. (2008). Social information foraging and sensemaking. In *Sensemaking Workshop*. Retrieved from https://www.researchgate.net/profile/Peter_Pirolli/publication/228845456_Social_Information_Foraging_and_Sensemaking/links/02bfe50f09ca794c52000000.pdf

Pirolli, P. L. (2009). An Elementary Social Information Foraging Model. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 605–614). New York, NY, USA: ACM. http://doi.org/10.1145/1518701.1518795

Pirolli, P. L., & Card, S. K. (1995). Information Foraging in Information Access Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 51–58). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. http://doi.org/10.1145/223904.223911

Pirolli, P. L., & Card, S. K. (1998). Information Foraging Models of Browsers for Very Large Document Spaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (pp. 83–93). New York, NY, USA: ACM. http://doi.org/10.1145/948496.948509

Pirolli, P. L., & Card, S. K. (1999). Information foraging. *Psychological Review*, *106*(4), 643.

Pirolli, P. L., & Card, S. K. (2005). The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis* (Vol. 5, pp. 2–4). Retrieved from https://www.e-education.psu.edu/geog885/sites/www.e-education.psu.edu.geog885/files/geog885q/file/Lesson_02/Sense_Making_206_Camera_Ready_Paper.pdf

Pirolli, P. L., Fu, W., Chi, E. H., & Farahat, A. (2005). Information scent and web navigation: Theory, models and automated usability evaluation. In *Proc. HCI International*. Retrieved from https://www.researchgate.net/profile/Peter_Pirolli/publication/228963831_Information_scent_and_web_navigation_Theory_models_and_automated_usability_evaluation/links/0912f50d23f3e9e9e6000000.pdf

Pirolli, P. L., & Fu, W.-T. (2003). SNIF-ACT: A Model of Information Foraging on the World Wide Web. In *Proceedings of the 9th International Conference on User Modeling* (pp. 45–54). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1759957.1759968

Pirolli, P. L., Pitkow, J., & Rao, R. (1996). Silk from a Sow's Ear: Extracting Usable Structures from the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 118–125). New York, NY, USA: ACM. http://doi.org/10.1145/238386.238450

Qi, D., Roychoudhury, A., Liang, Z., & Vaswani, K. (2012). DARWIN: An Approach to Debugging Evolving Programs. *ACM Trans. Softw. Eng. Methodol.*, *21*(3), 19:1–19:29. http://doi.org/10.1145/2211616.2211622

Reiss, S. P. (2009). Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 243–253). Washington, DC, USA: IEEE Computer Society. http://doi.org/10.1109/ICSE.2009.5070525

Ren, X., Shah, F., Tip, F., Ryder, B. G., & Chesley, O. (2004). Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (pp. 432–448). New York, NY, USA: ACM. http://doi.org/10.1145/1028976.1029012

Resnick, P., & Varian, H. R. (1997). Recommender Systems. *Commun. ACM*, *40*(3), 56–58. http://doi.org/10.1145/245108.245121

Robillard, M. P., & Chhetri, Y. B. (2015). Recommending reference API documentation. *Empirical Software Engineering*, *20*(6), 1558–1586. http://doi.org/10.1007/s10664-014-9323-y

Robillard, M. P., & Murphy, G. C. (2003). Automatically inferring concern code from program investigation activities. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings* (pp. 225–234). IEEE. http://doi.org/10.1109/ASE.2003.1240310

Roehm, T., Tiarks, R., Koschke, R., & Maalej, W. (2012). How Do Professional Developers Comprehend Software? In *Proceedings of the 34th International Conference on Software Engineering* (pp. 255–265). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337254

Romero, P., du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, *65*(12), 992–1009. http://doi.org/10.1016/j.ijhcs.2007.07.005

Safyallah, H., & Sartipi, K. (2006). Dynamic Analysis of Software Systems using Execution Pattern Mining. In *14th IEEE International Conference on Program Comprehension (ICPC'06)* (pp. 84–88). http://doi.org/10.1109/ICPC.2006.19

Savage, T., Revelle, M., & Poshyvanyk, D. (2010). FLAT3: Feature Location and Textual Tracing Tool. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2* (pp. 255–258). New York, NY, USA: ACM. http://doi.org/10.1145/1810295.1810345

Sawadsky, N., Murphy, G. C., & Jiresal, R. (2013). Reverb: Recommending Code-related Web Pages. In *Proceedings of the 35th International Conference on Software Engineering* (pp. 812–821). Piscataway, NJ, USA: IEEE. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486895

Scaffidi, C. (2010). Sharing, finding and reusing end-user code for reformatting and validating data. *Journal of Visual Languages & Computing*, *21*(4), 230–245. http://doi.org/10.1016/j.jvlc.2010.06.001

Schiller, T. W., Donohue, K., Coward, F., & Ernst, M. D. (2014). Case Studies and Tools for Contract Specifications. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 596–607). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568285

Schultze, S. J. (2002). A Collaborative Foraging Approach to Web Browsing Enrichment. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems* (pp. 860–861). New York, NY, USA: ACM. http://doi.org/10.1145/506443.506635

Schummer, T. (2001). Lost and found in software space. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001* (p. 10 pp.–). http://doi.org/10.1109/HICSS.2001.927261

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, *25*(4), 557–572. http://doi.org/10.1109/32.799955

Servant, F., & Jones, J. A. (2012). WhoseFault: Automatic Developer-to-fault Assignment Through Fault Localization. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 36–46). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337228

Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software*, *7*(6), 15–24. http://doi.org/10.1109/52.60586

Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., & Vijay-Shanker, K. (2007). Using Natural Language Program Analysis to Locate and Understand Action-oriented Concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development* (pp. 212–224). New York, NY, USA: ACM. http://doi.org/10.1145/1218563.1218587

Sillito, J., Murphy, G. C., & De Volder, K. (2006). Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 23–34). New York, NY, USA: ACM. http://doi.org/10.1145/1181775.1181779

Sillito, J., Voider, K. D., Fisher, B., & Murphy, G. C. (2005). Managing software change tasks: an exploratory study. In *2005 International Symposium on Empirical Software Engineering, 2005.* (p. 10 pp.–). IEEE. http://doi.org/10.1109/ISESE.2005.1541811

Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics based refactoring. In *Fifth European Conference on Software Maintenance and Reengineering, 2001* (pp. 30–38). http://doi.org/10.1109/.2001.914965

Singer, J., Elves, R., & Storey, M.-A. (2005). NavTracks: supporting navigation in software. In *13th International Workshop on Program Comprehension (IWPC'05)* (pp. 173–175). IEEE. http://doi.org/10.1109/WPC.2005.25

Sinha, V., Karger, D., & Miller, R. (2006). Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 187–194). IEEE. http://doi.org/10.1109/VLHCC.2006.40

Spool, J. M., Perfetti, C., & Brittan, D. (2004). *Designing for the scent of information*. User Interface Engineering.

Sridhara, G., Hill, E., Pollock, L., & Vijay-Shanker, K. (2008). Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *The 16th IEEE International Conference on Program Comprehension, 2008. ICPC 2008* (pp. 123–132). IEEE. http://doi.org/10.1109/ICPC.2008.18

Srinivasa Ragavan, S., Kuttal, S. K., Hill, C., Sarma, A., Piorkowski, D., & Burnett, M. (2016). Foraging Among an Overabundance of Similar Variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (pp. 3509–3521). New York, NY, USA: ACM. http://doi.org/10.1145/2858036.2858469

Stephens, D. W., & Krebs, J. R. (1986). Foraging theory: monographs in behavior and ecology. *Foraging Thoery: Monographs in Behavior and Ecology*.

Stolee, K. T., Elbaum, S., & Dobos, D. (2014). Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.*, *23*(3), 26:1–26:45. http://doi.org/10.1145/2581377

Storey, M.-A., Best, C., Michaud, J., Rayside, D., Litoiu, M., & Musen, M. (2002). SHriMP Views: An Interactive Environment for Information Visualization and Navigation. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems* (pp. 520–521). New York, NY, USA: ACM. http://doi.org/10.1145/506443.506459

Storey, M.-A., Cheng, L.-T., Singer, J., Muller, M., Myers, D., & Ryall, J. (2007). How Programmers can Turn Comments into Waypoints for Code Navigation. In *2007 IEEE International Conference on Software Maintenance* (pp. 265–274). http://doi.org/10.1109/ICSM.2007.4362639

Storey, M.-A., Ryall, J., Bull, R. I., Myers, D., & Singer, J. (2008). TODO or to bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 251–260). ACM/IEEE. http://doi.org/10.1145/1368088.1368123

Subramanian, S., Inozemtseva, L., & Holmes, R. (2014). Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 643–652). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568313

Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., & Su, Z. (2011). Synthesizing Method Sequences for High-coverage Testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (pp. 189–206). New York, NY, USA: ACM. http://doi.org/10.1145/2048066.2048083

Thung, F., Le, T.-D. B., Kochhar, P. S., & Lo, D. (2014). BugLocalizer: Integrated Tool Support for Bug Localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 767–770). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661678

Tokuda, L., & Batory, D. (2001). Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, *8*(1), 89–120. http://doi.org/10.1023/A:1008715808855

Toomim, M., Begel, A., & Graham, S. L. (2004). Managing Duplicated Code with Linked Editing. In *2004 IEEE Symposium on Visual Languages and Human Centric Computing* (pp. 173–180). http://doi.org/10.1109/VLHCC.2004.35

Treude, C., Barzilay, O., & Storey, M.-A. (2011). How do programmers ask and answer questions on the web?: NIER track. In *2011 33rd International Conference on Software Engineering (ICSE)* (pp. 804–807). http://doi.org/10.1145/1985793.1985907

Treude, C., & Storey, M.-A. (2010). Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (Vol. 1, pp. 365–374). http://doi.org/10.1145/1806799.1806854

Treude, C., & Storey, M.-A. (2011). Effective Communication of Software Development Knowledge Through Community Portals. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 91–101). New York, NY, USA: ACM. http://doi.org/10.1145/2025113.2025129

Treude, C., & Storey, M.-A. (2012). Work Item Tagging: Communicating Concerns in Collaborative Software Development. *IEEE Transactions on Software Engineering*, *38*(1), 19–34. http://doi.org/10.1109/TSE.2010.91

van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings* (pp. 97–106). http://doi.org/10.1109/WCRE.2002.1173068

Weiser, M. (1981). Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (pp. 439–449). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=800078.802557

Wiedenbeck, S., & Evans, N. J. (1986). BEACONS IN PROGRAM COMPREHENSION. *SIGCHI Bulletin*, *18*(2), 56–57. http://doi.org/10.1145/15683.1044090

Wilde, N., & Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, *7*(1), 49–62. http://doi.org/10.1002/smr.4360070105

Würsch, M., Ghezzi, G., Reif, G., & Gall, H. C. (2010). Supporting Developers with Natural Language Queries. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (pp. 165–174). New York, NY, USA: ACM. http://doi.org/10.1145/1806799.1806827

Xiao, L., Cai, Y., & Kazman, R. (2014). Titan: A Toolset That Connects Software Architecture with Quality Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 763–766). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661677

Xing, Z., & Stroulia, E. (2006). Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *2006 22nd IEEE International Conference on Software Maintenance* (pp. 458–468). http://doi.org/10.1109/ICSM.2006.52

Ye, Y., Fischer, G., & Reeves, B. (2000). Integrating Active Information Delivery and Reuse Repository Systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications* (pp. 60–68). New York, NY, USA: ACM. http://doi.org/10.1145/355045.355053

Ye, Y., Yamamoto, Y., & Nakakoji, K. (2007). A Socio-technical Framework for Supporting Programmers. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (pp. 351–360). New York, NY, USA: ACM. http://doi.org/10.1145/1287624.1287674

Yu, K., Lin, M., Chen, J., & Zhang, X. (2012). Practical isolation of failure-inducing changes for debugging regression faults. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 20–29). http://doi.org/10.1145/2351676.2351681

Zhang, S., & Ernst, M. D. (2014). Which Configuration Option Should I Change? In *Proceedings of the 36th International Conference on Software Engineering* (pp. 152–163). New York, NY, USA: ACM. http://doi.org/10.1145/2568225.2568251

Zhang, T., Song, M., & Kim, M. (2014). Critics: An Interactive Code Review Tool for Searching and Inspecting Systematic Changes. In *Proceedings of the 22Nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering (pp. 755–758). New York, NY, USA: ACM. http://doi.org/10.1145/2635868.2661675

Zhang, Y., & Hou, D. (2013). Extracting problematic API features from forum discussions. In *2013 21st International Conference on Program Comprehension (ICPC)* (pp. 142–151). http://doi.org/10.1109/ICPC.2013.6613842

Zhao, W., Zhang, L., Liu, Y., Sun, J., & Yang, F. (2004). SNIAFL: towards a static non-interactive approach to feature location. In *26th International Conference on Software Engineering, 2004. ICSE 2004. Proceedings* (pp. 293–303). http://doi.org/10.1109/ICSE.2004.1317452

Zimmermann, T., Zeller, A., Weissgerber, P., & Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, *31*(6), 429–445. http://doi.org/10.1109/TSE.2005.72