Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance?

Joseph Lawrance
Oregon State University
School of EECS
Corvallis, Oregon 97331
lawrance@cs.orst.edu

Rachel Bellamy
IBM T.J. Watson Research
19 Skyline Drive
Hawthorne, New York 10532
rachel@us.ibm.com

Margaret Burnett
Oregon State University
School of EECS
Corvallis, Oregon 97331
burnett@cs.orst.edu

Abstract

During maintenance, professional developers generate and test many hypotheses about program behavior, but they also spend much of their time navigating among classes and methods. Little is known, however, about how professional developers navigate source code and the extent to which their hypotheses relate to their navigation. A lack of understanding of these issues is a barrier to tools aiming to reduce the large fraction of time developers spend navigating source code. In this paper, we report on a study that makes use of information foraging theory to investigate how professional developers navigate source code during maintenance. Our results showed that information foraging theory was a significant predictor of the developers' maintenance behavior, and suggest how tools used during maintenance can build upon this result, simply by adding word analysis to their reasoning systems.

1. Introduction

Recent research has shown that programmers spend up to 35% of their time navigating source code [8, 12]. Much code navigation done by programmers occurs when they are scanning source code trying to find the pieces that are important for their current task [8]. Understanding more about *how* programmers navigate can inform tools that facilitate code navigation during maintenance.

There is a long history of research in program debugging (e.g., [4, 7, 20]), a task which is central to maintenance. This work has resulted in a general consensus that developers, one way or another, develop hypotheses, and then follow these hypotheses to perform their tasks.

In this paper, we explore the proposition that these hypotheses have linguistic relationships to the words with which a bug or feature request was originally described, and further that these linguistic relationships can help determine the terms they will search for in the source code and to which files they will navigate.

In essence, we are interested in the applicability of information foraging theory to program maintenance. Information foraging theory has emerged in the last decade as a way to explain how people seek, gather, and make use of information [17]. It is based on optimal foraging theory, a theory of how predators and prey behave in the wild. In the domain of information technology, the predator is the person in need of information and the prey is the information itself. The predator/prey model, when translated to the information technology domain, has been shown to mathematically model which web pages human information foragers select on the web [15], and therefore has become extremely useful as a practical tool for web site design and evaluation [5, 14, 18].

Can information foraging theory also be used as the basis for tools to support navigation in maintenance? That is, is much of maintenance a foraging task, in which programmers follow scent in order to navigate to patches of code?

In this paper, we describe a study of professional programmers maintaining a real-world program. We studied two types of maintenance situations. The first was finding and fixing the part of the program not working correctly (a bug). The second was adding missing functionality, which involved finding a sensible place in the code (a hook) at which to insert new code that can provide this functionality. Our study investigates whether techniques used in information foraging theory can model programmers' behavior in these two situations.

2. Background and Related Work

In Ko et al.'s investigation of developer behavior during software maintenance, the participants—student developers working in Eclipse with nine source files—spent 35% of their time navigating source code [12]. This surprising result made clear the importance of

0-7695-2987-9/07 $25.00 © 2007 IEEE
DOI 10.1109/VLHCC.2007.25

trying to understand how programmers go about navigating, and how to help them save time while doing so. This finding led to the development of a model of program understanding [12], which proposes that the cues (e.g., identifier names, comments, and documentation) in an environment are central to searching, relating, and navigating code in software maintenance and debugging. Although they did not investigate scent per se, their model is philosophically similar to information foraging theory.

DeLine et al. [8] conducted empirical work into problems arising in professional developers' navigations through unfamiliar code. Their work turned up two major problems: developers needed to scan a lot of the source code to find the important pieces (echoing the finding of [12]), and they tended to get lost while exploring. These results inspired the idea to combine collaborative filtering and computational "wear" from users' interaction histories into a concept called "wear-based filtering." They developed this idea in their Team Tracks system [8] with the goal of easing program comprehension and reducing the cost of navigation by showing the source code navigations of fellow development team members. Team Tracks shows related items to the selected one (most frequently visited before/after), and favorite classes (collectively navigated to most often). Results from both a laboratory and a field study showed that sharing navigation data helped ease program comprehension and navigation.

The Hipikat system [6] also provides search for code-related artifacts. It is based on the notion of relationships among artifacts, which are derived through "project memory," which remembers paths traversed by earlier members of a team, and textual similarity. The textual similarity part is a hand-crafted textual similarity matcher that assigns weights to words based on global prevalence of the word in the repository and local prevalence of the word to the document. Evaluations showed that Hipikat was able to provide a useful pointer to relevant files, and further that newcomers can use the information presented by Hipikat to achieve results comparable in quality and correctness to those of more experienced members of the team.

Regarding textual analysis of the words in bug reports themselves, [11] analyzed 200,000+ Open Source bug report titles. They were able to ascertain patterns in the structure of these titles, with implications for how to automatically tell different kinds of bugs apart. Their findings most relevant to our work included the fact that 54% of nouns were proper nouns referring to code, project, or file entities. Nine common verbs played a grammatical mood role, but other verbs often had an identification role, such as identifying computational operations (e.g., "open"). Adverbs, adjectives, and prepositions did not directly identify entities.

Information foraging theory has a strong relationship with textual analysis. It makes its predictions based on proximal cues associated with links. Proximal cues are, as the term implies, cues that are near a link. Examples include link text, URL name, and surrounding text and graphics. Cues serve as scent as to where the prey might be hiding. Diet is also important in foraging: an animal would like to get the most nutrients for as little effort as possible. Because some locations (patches) have a greater density of prey than others, animals use cues such as the strength of the scent of the prey to decide where to spend their time.

Information scent is calculated by analysis of the words in the text surrounding a link in the SNIF-ACT information foraging model [16]. The information foraging computations are based on the ACT spreading activation cognitive architecture [1]. In spreading activation models, the content of long-term human memory (LTM) is modeled by a network of nodes, where each node is a chunk of information, and links represent associations between information chunks. Chunks that have been experienced together in the past have stronger associations than chunks that have not commonly co-occurred. At any point in time, a chunk has a certain level of activation; the more recently experienced, the greater the activation. A chunk's activation level at the first point in time of interest is referred to as its base-level activation.

When the hypothetical user modeled by SNIF-ACT looks at a web page, the page contents activate nodes in the network. This activation spreads to related nodes, spreading more to nodes that have a strong association. Thus, the final activation of a chunk is determined by its base-level activation, plus the activation it receives from chunks associated with it. At any point in time, the hypothetical user's working memory comprises active nodes in LTM, together with a representation of their current information need. The content of working memory then determines which actions will be selected. Thus, the hypothetical user will choose to click on links on the web page whose proximal cues correspond to both the currently most active nodes in LTM, and to their current information need. If however, the activation due to the web page is less than a certain threshold, the user may choose to leave that information patch altogether.

This spreading activation model has been shown to model the actual search behavior of web users [16], showing that linguistic knowledge and analysis of linguistic relatedness and salience are fundamental to how users navigate web information.

3. Empirical Study

The goal of this study was to ascertain whether in-

formation foraging theory could be applied to the prediction and understanding of developers' behaviors during the maintenance activities of fixing a bug and adding new functionality.

3.1 Design, Participants, and Materials

We recruited 12 professional programmers from IBM. We required that each have at least two years experience programming Java, used Java for the majority of their software development, were familiar with Eclipse and bug tracking tools, and felt comfortable with searching, browsing, and finding bugs in code for a 3-hour period of time.

We searched for a program that met several criteria: we needed access to the source code, it needed to be written in Java, and it needed to be editable and executable through Eclipse, a standard Java IDE. We selected RSSOwl, an open source news reader that is one of the most actively maintained and downloaded projects hosted at Sourceforge.net. The popularity of newsreaders and the similarity of its UI to email clients meant that our participants would understand the functionality and interface after a brief introduction, ensuring that our participants could begin using and testing the program immediately.

RSSOwl (Figure 1) consists of three main panels: to the left, users may select news feeds from their favorites, to the upper right, users can review newsfeed headlines. On selecting a headline, the story appears in the lower right panel of the application window.

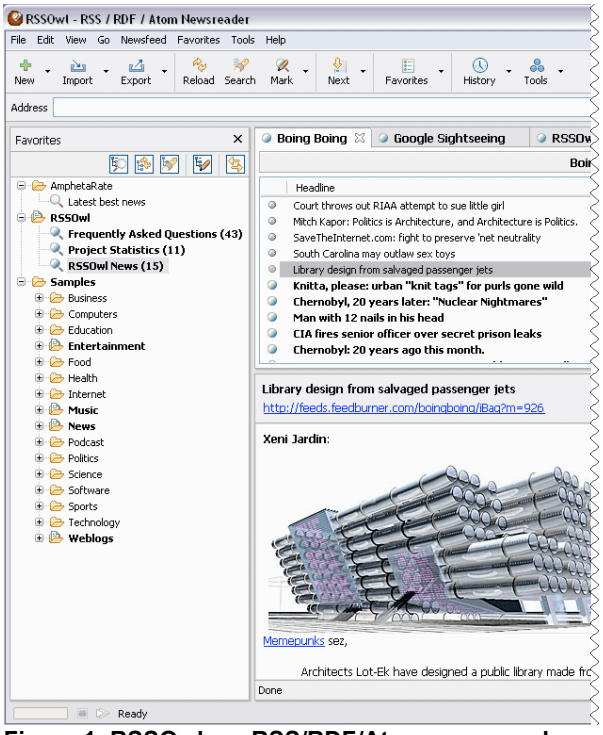

Figure 1. RSSOwl, an RSS/RDF/Atom news reader.

Having decided upon the program, we also needed issues (bug reports) for our participants to work on during the study. Since we were interested in source code navigation and not the actual bug fixes, we wanted to ensure that the issue could not be solved within the duration of the session. We also decided that one issue should be about fixing erroneous code and the other about providing a missing feature. From these requirements, we selected two issues: 1458101: "HTML entities in titles of atom items not decoded" and 1398345: "Remove Feed Items Based on Age." We will refer to the first as issue B ("Bug") and the second as issue MF ("Missing Feature"). Each participant worked on both issues, and we counterbalanced the ordering of issues among subjects to control for learning effects. The former involves finding and fixing a bug, and the latter involves inserting missing functionality, requiring the search for a hook.

The issues we assigned to developers were open issues in RSSOwl. We considered looking at closed issues whose solution we could examine, but this would have meant locating an older version of RSSOwl for participants to work on, and would have required us to ensure that participants would not find the solution accidentally by browsing the web. Therefore, we decided that our participants would work on open issues, cognizant of the risk that RSSOwl's own developers could close the issues during the study, updating the web-available solution with the correct code in the process. (Fortunately, this did not happen.)

3.2. Procedure

We observed each participant for three hours. Upon their arrival, after participants filled out initial paper work, we briefly described what RSSOwl is, and explained to our participants that we wanted them to try to find and possibly fix issues that we assigned to them. We then set up the instant messenger so that participants could contact us remotely. Then we excused ourselves from the room.

We recorded electronic transcripts and video of each session using Morae screen and event log capture software. We archived the changes they made, if any. The electronic transcripts and source code served as the data sources we used in our analysis.

4. Results

Information foraging theory consists of three major components: information patches, information scent, and information diet. In addition, there is the concept of the prey itself, namely the ultimate information goal.

In adapting information foraging theory to maintenance tasks, we assume that the *prey* is the place(s) in the source code where the corrections need to be made. Information *patches* are the places in the source code in which the prey might hide. We begin by considering patches, and then investigate whether the prey's *scent* can be approximated linguistically, and whether the notion of *diet*, which pertains to perceived profitability of following scents to one patch instead of another, appears to be present.

4.1 A Prerequisite: Patches

Do information foraging theory's patches apply to maintenance? Narrowing down the search is part of maintenance, and conceptually, this seems to be the same thing as defining the relevant information patch.

To see whether this intuitive notion of patches is indeed consistent with maintenance, we will take the view that patches can be considered to consist of (Java) classes. Information foraging theory should predict that participants would spend most of their time visiting only a small fraction of the patches available to them.

Participants' individual visits were indeed consistent with this prediction. Participants concentrated their attention on a tiny fraction of the classes available in RSSOwl. Out of 193 classes in RSSOwl, participants examined an average of only 5% and 9% of the classes for issues B and MF, respectively, with a standard deviation of 3% and 6%, respectively.

Consensus results were also consistent with the notion of patches. We counted how many participants visited the same classes as a measure of consensus (Table 1). Depending on where the threshold is set as to how many participants must agree for consensus, the agreed-upon patches consisted of 2 to 42 of the classes (1% to 22% of the total). For example, for issue MF, the same two classes were visited by 8 of the participants. For issue B and MF, 144 and 100 classes, respectively, were never touched by any participant. Given these individual and consensus results, it is clear that the notion of patches at the granularity of classes was consistent with maintenance behavior.

≥N participants viewed the same classes:	Classes viewed for Issue B	Classes viewed for Issue MF
≥6 participants	2 classes (1%) (NewsItem, StringShop)	5 classes (3%) (GUI, RSSOwl-Loader, NewsItem, Channel, ArchiveManager)
≥5 participants	7 classes (4%)	9 classes (5%)
≥4 participants	11 classes (6%)	17 classes (9%)
≥3 participants	15 classes (8%)	23 classes (12%)
≥2 participants	24 classes (12%)	42 classes (22%)

Table 1. Degree of consensus in exactly where the source code "patches" were. Counts are of the *same* classes visited by N participants.

Note that the agreed-upon patch size was smaller for issue B than for issue MF (Mann-Whitney, $U=1344.5$, $p < 0.001$). It is possible that this is because our feature request could be implemented in a greater variety of ways and in a greater variety of locations, which was the case for participants' handling of the issues in this study.

We also did a cluster analysis to see if the source code words would cluster related classes together, and thus possibly be an indicator of the scent of the patch. The result was that classes did indeed cluster into groups based on the functionality they implement. The dendogram in Figure 2 points out several examples.

Having satisfied ourselves as to the prerequisite notion of patches being consistent with the navigation behavior we observed, we proceeded to the questions of scent and diet.

4.2 Information Scent

Given the presence of a few patches that so many participants agreed were worth visiting, the next question is whether the information scent, computed from words in the bug reports, leads to those patches:

RQ 1: Do the words in bug reports predict to which source code classes (patches) developers navigate?

We began with the words in the bug reports. We filtered out all words except nouns and verbs, based on the findings of Ko et al.'s linguistic analysis of bug

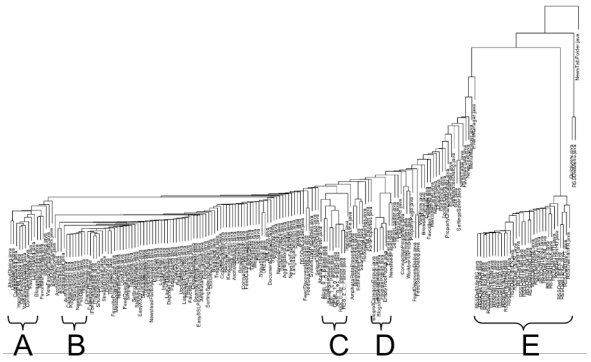

Figure 2. Dendogram of patches grouped by words. For example, "A" is the set of dialog box classes, "B" the sorting classes, "C" the parsing classes, "D" the popups, and "E" the internationalization.

report titles [11], which revealed that identification words were the nouns and verbs. Ko et al. also pointed to nine common verbs that did not contain identifying information (e.g., "is"), so we filtered out those nine as well. After filtering, we then computed how well the words in each bug report predicted where participants would navigate.

We computed participants' navigation choices in two ways. In our first comparison, the dependent variable was the total number of visits made to each of the 193 classes. In our second comparison, the dependent variable was participant consensus, the extent to which scent predicted the most popular source files in terms of the number of participants who visited them.

Interword correlation was the basis of both calculations, just as in Pirolli's information foraging calculations of interword correlation [15]. Thus, terms in documents were weighted according to the term-frequency inverse document frequency (TF-IDF) formula, commonly used in information retrieval systems, shown below [2].

$$w_{i,j} = f_{i,j} \times idf_i$$

$$\text{where } f_{i,j} = \frac{freq_{i,j}}{\max_{\forall v} freq_{v,j}} \quad \text{and} \quad idf_i = \log \frac{N}{n_i}$$

Here, $f_{i,j}$ is the frequency of word i in document j (normalized with respect to the most frequently occurring word v in a document), idf_i is the inverse document frequency, and $w_{i,j}$ is the weight of word i in document d_j. The interword correlation between source files and each bug report was then computed via cosine similarity (also commonly used in information retrieval systems), as shown below:

$$sim(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}}$$

where $w_{i,q}$ is the weight of word i in the query (bug report).

For predictive questions such as RQ1, linear regression is the appropriate test. The interword correlation between nouns and verbs in the bug report and words in each class was a significant predictor of participants' visits to those classes ($p < .001$). It was also a significant predictor of the consensus among developers as to which classes were relevant ($p < .001$). See Table 2. We also tested weighting class names higher than other words, given that most information retrieval systems assign higher weights to documents that have query key words in their titles, but we found that doing so did not affect the predictive power of the model.

These results suggest that source files were being selected at least in part on the basis of the words they contained relative to the words in the bug reports.

4.3 Diet

For information foraging theory, a central problem in information sense-making is the allocation of attention. This comes out especially in the notion of diet.

The information foraging notion of *diet* is that there is a principle of *lost opportunity,* which states that by handling lower-ranked items in the diet, one would lose the opportunity to go after higher-ranked items. The diet model says that, for this reason, people will pursue the most profitable items (gain from information per amount of time to deal with the information). Diet in information foraging theory has much in common with Blackwell's model of attention investment [3], which posits that people weigh costs, benefits, and risks of choosing which computer features to use and how to spend the time and attention available for the task.

Diet focuses on profitability. We emphasize that the aspect of interest is *perceived* profitability, which may not be true profitability. Thus, we must turn to our participants for where they perceived highest profitability to lie.

So far, we have shown that participants were likely to visit patches where the scent, originating in the bug report's words, took them. The research question in this section asks whether those files were the profitable places to visit, from the perspective of the participants. If they spent significant amounts of time in a file, we will take that to mean that this was indeed a place they deemed worthy of their attention.

RQ2: Do the words in bug reports predict the proportion of total time developers will spend in a particular file?

For this research question, we were interested in determining the extent to which interword correlation predicted where participants would allocate their time. As shown in Table 3, the interword correlation among bug reports and source files explained 27% to 31% of how the participants allocated their time among

Explanatory → Response	Issue B	Issue MF
Interword correlation → Visits	$R^2 = 0.287$, $F(1,191) = 78.17$, $p < 0.001$	$R^2 = 0.311$, $F(1,191) = 87.45$, $p < 0.001$
Interword correlation → Consensus	$R^2 = 0.267$, $F(1,191) = 70.9$, $p < 0.001$	$R^2 = 0.309$, $F(1,191) = 87.03$, $p < 0.001$

Table 2: Interword correlation as predictor of visits and consensus using linear regression.

classes. Figure 3 demonstrates the extent to which they focused on just a few classes.

4.4 Information Scent and Hypotheses

Did the words in bug reports predict where participants were *trying* to go? For example, they may have been visiting and spending time in certain files in a quest for information they did not find there.

Prior research into debugging suggests that programmers form hypotheses about the reasons and places relevant to the bugs, and that much of debugging revolves around attempts to confirm, refine, or refute those hypotheses [4, 10, 20].

Hypotheses about places were sometimes expressed explicitly, i.e., when participants typed in query strings. We decided to consider evidence of where they were trying to go by considering the contents of their query strings. Thus, we measured whether the words in bug reports predict participants' search query words.

RQ3: Do the words in bug reports predict the vocabulary of the queries developers made?

For both issues, the most commonly used words found in queries were indeed found in the words of the bug reports. Using the words in the bug report as a model of the words participants used in their queries, we found that for both issues, the bug report was predictive of the query words. (Issue B: $F(1,98) = 6.669$, $R^2 = 0.05$, $p = 0.0113$. Issue MF: $F(1,78) = 32.49$, $R^2 = 0.285$, $p < 0.001$.) Even though there was a significant relationship between the words in the bug reports and the queries in both issues, in the case of issue B, the relatively low R^2 value shows that there were many other factors at work as well. In the case of issue MF, the predictive value was moderately strong.

It surprised us that the vocabulary of the feature request predicted the words in searches for the new feature hook better than the vocabulary of the bug report predicted words in searches for the buggy code, given that, as discussed in section 4.1, participants reached consensus on fewer classes for the bug report. Further analysis revealed that participants found suitable hooks (GUI.java, RSSOwlLoader.java, etc.) using fewer queries and a more limited vocabulary than used in pursuing the bug. Once this hook was found, participants diverged in their subsequent navigation in pursuit of different ways to implement or address the feature request.

Explanatory → Response	Issue B	Issue MF
Interword correlation → Time span	$R^2 = 0.27$, $F(1,191) = 72.53$, $p < 0.001$	$R^2 = 0.310$, $F(1,191) = 87.21$, $p < 0.001$

Table 3: Interword correlation as predictor of participants' allocation of time among classes using linear regression.

Essentially, we are proposing that the hypotheses that the participants generated were determined not only by their past experience with how programming functionality is implemented, but also by the words that were used in the bug reports. That is, their hypotheses were in part shaped by the vocabulary choices of the bug reports. Similar effects of problem description on problem-solving strategy have been found in other problem solving domains [9].

5. Discussion

The goal of this experiment has been to contribute to prediction of developers' behavior, and the results showed significant relationships between word-based predictions and the participants' actual behavior. However, as the R-squared values show, the linguistic predictions did not explain all of their behavior.

R-squared values can range from 0 to 1. In studies of human behavior, R-squared values are common in the .09 to .25 range [13]. With one exception (.05), our R-squared values ranged from .26 to .31. Such R-squared values indicate, of course, that a great deal of the humans' behavior was not accounted for. That is to be expected because, first, there is inherent measurement error in attempting to measure human behavior and, second, it is not realistic to expect a single variable to account for all of their behavior.

Even so, the significant predictive relationship word analysis had with participants' navigation has useful implications for the design of future tools, as we discuss in the next section. In addition, these results provide independent evidence about the premises behind current systems like Team Tracks and Hipikat. The designers of these systems have conducted studies, but our study is independent, not conducted by the designers of Hipikat or Team Tracks.

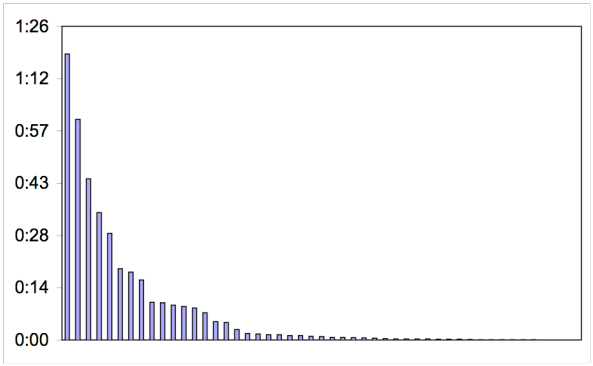

Figure 3: Total time spent in all 49 classes visited by anyone for Issue B. (Many are too low to register.) The steep drop-off in time spent shows the intense focus on just a few highly ranked classes.

Specifically, the results showed that the bug reports predicted the vocabulary of the queries participants used to find appropriate source code. This supports the concept of including textual analysis, used in systems like Hipikat. The R-squared values also showed that word analysis alone was unlikely to be enough, supporting the concept behind both Hipikat and Team Tracks that multiple sources of information are needed to make good navigation predictions, and that a single source is unlikely to have high enough accuracy.

Both Hipikat and Team Tracks make use of collective knowledge through approaches such as navigation path "wear" and class popularity. Our results support this design choice too, showing that participants collectively narrowed their focus on the same files. But given that participants reached more consensus for the bug than for the feature request, such systems may be able to improve further by factoring in which type of issue (bug or feature request) a developer is working on.

Turning to theoretical underpinnings, our results are consistent with a number of existing theories, and shed further light upon the way developers go about debugging and maintenance.

First, our results are consistent with the well-established idea of hypothesis formation as a basis of debugging [4, 10, 20]. In our results, vocabulary in the bug reports predicted vocabulary in the queries, suggesting the possibility of queries as a surrogate for hypotheses about "subject areas" in the source code where work will be needed. Further, the places to which participants navigated as a result of their queries were the "right" places for investigating their hypotheses, as evidenced by the fact that they spent significant time in the files once they had gotten there. Note also that vocabulary did not explain all of participants' navigation behavior, which is consistent with the idea of hypothesis formation as well, since it is unlikely that *all* of the vocabulary of suitable hypotheses would be present in a bug report.

Second, our results are consistent with beacons [19] and with Ko et al.'s model of searching, relating, and collecting [12]. The work on beacons emphasizes the importance of cues in comprehension, though beacons are at a finer granularity than what we examined. Ko et al.'s model is related to information foraging theory in its emphasis on the importance of cues in the environment. Our results further refine what goes on in the "search" aspect that leads developers to the right places for relating and collecting.

The consistency of our findings with information foraging theory itself has interesting implications. Namely, it points to the possibility of a cognitive model that can be used to understand program navigation. Just as Pirolli et al. [15] extended their initial static analysis of web navigation into the dynamic SNIF-ACT model, we can use the static analyses presented here as the basis for creating a dynamic model of scent-following behavior in maintenance. Developing such a model in the context of programming may promote understanding of how well a particular piece of code and/or bug report will enable foraging. It could also be used as the basis for tools to help programmers navigate more optimally, as we discuss next.

6. Practical Implications

Being able to predict programmers' navigation from analysis of the interword correlation of words in the bug reports and words in class files and queries has practical implications in at least three areas: for tools to help programmers debug, in helping programmers and testers write better bug reports, and in analyzing the usefulness of programmers' word choices in their code.

Regarding uses in tools, navigation tools could add analysis of the interword correlation between words in bug reports and source code to suggest classes that are most pertinent to the bug report. This may be as simple as providing a list of classes ordered by interword correlation, or could go further. For example, the linguistic analysis could contribute other mechanisms for reasoning about navigation recommendations, such as Team Tracks' devices. Fault localization tools could also benefit. Fault localization tools make their best guess as to where the bugs in source code may be lurking. These guesses often make use of multiple sources of information. Adding word analysis as another source of information could improve their ability to pinpoint the fault.

Some bug reports are better than others in enabling a programmer to find the bug's location. Our study suggests that the use of nouns and verbs that are the same as (or strongly associated with) words in the class files will help the programmer find source code locations relevant to the bug. This could be used as the basis of a usability analysis of bug reports, analogous to the information-foraging-based Bloodhound usability analysis tool [5] for web sites.

In a similar manner, analysis of the scent emanating from the source files could be used to help programmers write more navigable source code. Scent depends on what is considered to be the prey. Existing documentation, such as requirements or documents describing the application domain, could be used to define the different possible prey, and interword correlation between the documentation and the source code would provide an assessment of scent in the source code relevant to that prey. One possible use could be to automatically assess names a programmer chooses for classes, methods, and variables when writing code, or even the word choices in his or her comments.

We have mentioned the potential of word analysis for contributing to existing tools' sources of information. The other direction is possible too: these sources of information could contribute to word analysis. For example, historic data about which class files were modified in order to close which bug report could provide application-specific insights into word relationships. These insights could be used to weight association strength between words that co-occur in that application's class files and its bug reports.

The implications discussed above draw on the specific word analysis techniques used in our study. Taken more generally, we anticipate that information foraging theory will have an even broader set of implications concerning how to increase the scent of documentation and code for different programming tasks, offering a wealth of design implications for programming artifacts and tools.

7. Conclusion

In this study, we considered, from the perspective of information foraging theory, the linguistic relationships between hypotheses, words in two particular bug reports, and professional developers' navigation through source code as they attempted to fix the bugs.

We found that the central tenet of information foraging theory, that of following scent to patches, was a significant predictor of our participants' navigation choices. This was true for both of the bugs we considered; that is, it occurred both in searching for a bug in source code and in searching for a place to "hook" new code to support a missing feature. Specifically, we found statistical evidence of the three components of information foraging theory: of patches, of following scent, and of the pursuit of profitability (diet). We also found statistical ties between words in bug reports and participants' queries, a partial approximation of their hypotheses about locations for fixing the bugs.

This study is far from the last word, and there are many possibilities for follow-up research and refinement. Still, the practical implications are clear: the results strongly suggest how tools used in maintenance can make use of these findings, simply by adding word analysis to their reasoning systems.

Acknowledgments

This study was supported in part by IBM Research, by the EUSES Consortium via NSF ITR-0325273, and by an IBM International Faculty Award.

References

[1] Anderson, J. R., Bothell, D., Byrne, M., Douglass, D., Lebiere, C. and Qin, Y. An integrated theory of mind. *Psychological Review* 111(4), 2004, 1036-1060.

[2] Baeza-Yates, R., Ribeiro-Neto, B. *Modern Information Retrieval*, Addison Wesley Longman, 1999.

[3] Blackwell, A. First steps in programming: A rationale for attention investment models, *IEEE Symp. Human-Centric Comp. Langs. Envs.*, 2002, 2-10.

[4] Brooks, R. Towards a theory of the cognitive processes in computing programming, *Int. J. Human-Computer Studies* 51, 1999, 197-211.

[5] Chi, E., Rosien, A., Supattanasiri, G., Williams, A., Royer, C., Chow, C., Robles, E., Dalal, B., Chen, J., Cousins, S. The Bloodhound project: Automating discovery of web usability issues using the InfoScent simulator, *ACM Conf. Human Factors Comp. Sys.*, Ft. Lauderdale, Florida, 2003.

[6] Cubranic, D., Murphy, G., Singer, J., Booth, K. Hipikat: A project memory for software development, *IEEE Trans. Software Engineering* 31(6), 446-465, June 2005.

[7] Detienne, F., *Software Design - Cognitive Aspects*, Springer, 2001.

[8] DeLine, R., Czerwinski, M. and Robertson, G., Easing program comprehension by sharing navigation data, *IEEE Symp. Visual Langs. Human Centric Computing*, 2005, 241-248.

[9] Glick, M. and Holyoak, K. J., Schema induction and analogical transfer. *Cognitive Psychology* 1983, 15, 1-38.

[10] Ko, A., Myers, B., A framework and methodology for studying the causes of software errors in programming systems, *J. Visual Langs. Computing* 16(1-2), 2005.

[11] Ko, A., Myers, B., and Chau, D., A linguistic analysis of how people describe software problems, *IEEE Symp. Visual Langs. Human-Centric Computing*, 2006, 127-136.

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

[13] Mitchell, M. and Jolley, J. *Research Design Explained*, 6th Ed., Wadsworth Publishing, 2006.

[14] Nielsen, J. Information foraging: Why Google makes people leave your site faster http://www.useit.com/alertbox/20030630.html. (June 30, 2003.)

[15] Pirolli, P. Computational models of information scent-following in a very large browsable text collection. *ACM Conf. Human Factors Comp. Sys.*, 1997, 3-10.

[16] Pirolli, P. and Fu, W.-T., SNIF-ACT: A model of information foraging on the World Wide Web. *Lecture Notes in Computer Science* 2702, 2003, 45-54.

[17] Pirolli, P. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, 2007.

[18] Spool, J., Profetti, C., and Britain, D., Designing for the scent of information, *User Interface Engineering*. 2004.

[19] Wiedenbeck, S., Scholz, J. Beacons: A knowledge structure in program comprehension, in *Designing and Using Human-Computer Interfaces and Knowledge Based Systems* (G. Salvendy, M. Smith, ed.), Elsevier, 1989, 82-87.

[20] Vans, A. and von Mayrhauser, A., Program understanding behavior during corrective maintenance of large-scale software, *Int'l J. Human-Computer Studies* 51(1), 1999.