Information Foraging as a Foundation for Code Navigation (NIER Track)

Nan Niu
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
niu@cse.msstate.edu

Anas Mahmoud
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
amm560@msstate.edu

Gary Bradshaw
Department of Psychology
Mississippi State University
Mississippi State, MS 39762
glb2@ra.msstate.edu

ABSTRACT
A major software engineering challenge is to understand the fundamental mechanisms that underlie the developer's code navigation behavior. We propose a novel and unified theory based on the premise that we can study developer's information seeking strategies in light of the foraging principles that evolved to help our animal ancestors to find food. Our preliminary study on code navigation graphs suggests that the tenets of information foraging provide valuable insight into software maintenance. Our research opens the avenue towards the development of ecologically valid tool support to augment developers' code search skills.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques.

General Terms
Design, Human Factors.

Keywords
Software maintenance, program comprehension, foraging theory.

1. INTRODUCTION
Program comprehension is a key developer activity during software maintenance. Before attempting any modification, developers must navigate, locate, and understand the parts of the software system relevant to the desired change. Because the code fragments involved in a software maintenance task are typically distributed throughout multiple system modules, code navigation can be both time-consuming and difficult [1]. Therefore, a major challenge in software engineering research is to understand the fundamental mechanisms that underlie the developers' code navigation behaviors and further develop tool support to augment their information seeking skills during software maintenance.

Current code navigation research falls mainly into two categories: *ad hoc* tools developed without any underlying theoretical basis and derivation of descriptive theories from observing developer behavior. What is not known is to what extent a *unified* theory can explain and predict developers' behaviors by taking their maintenance goals and the source code environment into consideration. Lack of such knowledge is an important problem, because *ad hoc* tool building can cover the fundamental principles only partially and the many descriptive theories are already too specific to be generalizable.

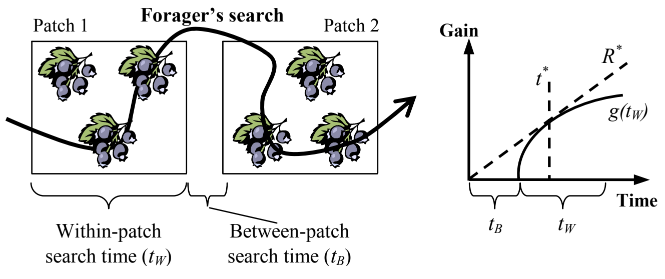
In this paper, we propose to develop a unified code navigation theory based on *information foraging* whose validity has been empirically confirmed in the domain of Web navigation [2]. The appeal of developing a *unified* theory is that a set of general assumptions can both account for all descriptive models and guide tool builders toward principled ways to increase practical support for developers. We hypothesize that developers' search for relevant code can be mathematically modeled in terms of the "built-in" foraging mechanisms that evolved to help our animal ancestors to find food and help users to find useful information on the Web. In other words, we contend that developers are well adapted to the plethora of information in the code space, and that they have evolved the strategies to maximize the gains of useful information to their tasks per unit cost.

Only recently has information foraging theory been applied to code navigation. Ko *et al.* were among the first to relate foraging theory to developers' seeking relevant code in maintenance [3]. Lawrance and colleagues mapped foraging theory's constructs to the debugging domain in a series of studies [4-6] and presented encouraging results that matched the theory's predictions with the developers' actual navigations. Even these efforts have not examined the assumptions of the foraging model to determine whether or not they are met in program comprehension.

This paper aims to shed light on the scope of information foraging theory's applicability in software engineering. Section 2 traces the theory's root to optimal food foraging and reviews its recent extension to debugging. Section 3 presents our vision of a unified code navigation theory. Section 4 describes our ongoing efforts in realizing the vision by highlighting a preliminary study on code navigation graphs. Section 5 concludes the paper and overviews challenges and potential applications of information foraging in software maintenance.

2. BACKGROUND AND RELATED WORK
Information foraging theory was originally inspired by appeals in the psychology literature for an ecological approach to understanding human information-gathering and sense-making strategies [2]. The general idea is that we can scientifically study human and technological adaptations to the flux of information in the social environment in much the same manner as biological adaptations to the flux of energy in the physical environment.

Information foraging derives from optimal foraging theory in biology and anthropology, which analyzes the adaptive value of food-foraging strategies [7]. A key assumption is that animals (including humans) should have well-designed food-seeking strategies because higher rates of energy consumption should generally translate in higher reproductive success. Central to optimal foraging theory are the *patch* model and the *diet* model.



(a) Patchy environment (b) Charnov's Theorem [8]

Figure 1. The patch model

The patch model deals with predictions of the amount of time an organism would forage in a patch before leaving for another patch. Figure 1a illustrates the model by presenting a hypothetical bird foraging in an environment that consists of patches of berry clusters. The forager must expend some amount of *between-patch* time (t_B) getting to the next patch. Once in a patch, the forager faces the decision of keeping *within-patch* foraging (t_W) or leaving to seek a new patch. As the forager gains energy, the amount of food diminishes or depletes. In such cases, there will be a point at which the expected future gains from foraging within a current patch diminish to the point that they are less than the expected gains that could be made by leaving for a new one. Figure 1b shows Charnov's Theorem [7], which mathematically models an optimal forager's time allocation. In Figure 1b, $g(t_W)$ represents a decelerating expected net gain function. The amount of energy gained per unit time of foraging is $R = g(t_W) / (t_B + t_W)$. Thus, the optimal time to spend in patch, t^*, occurs when the derivative of $g(t_W)$ is equal to the slope of the tangent line R^*.

The diet model deals with the tradeoffs when a predator forages in a habitat that contains a variety of prey. If a predator's diet is too narrow (e.g., it eats only a few types of prey), it will spend all of its time searching. If the predator's diet is too broad (e.g., it eats every type that encountered), then it will pursue too much unprofitable prey. Optimal diet selection algorithms are based on two principles [7]. The *profitability* principle states that the prey is predicted to be ignored if its profitability, $\pi = g/t_W$, is less than the expected rate of gain, R, of continuing search for other types of prey. The *prevalence* principle states that increases in higher profitability prey's prevalence (i.e., encounter rate), $\lambda = 1/t_B$, make it optimal to be more selective. Although the patch model and the diet model assume that the forager has "global" information concerning the environment, the models are elementary building blocks of optimal foraging theory and have generally proven to be productive and resilient in addressing food-foraging behaviors studied in the field and the lab [7].

Pirolli [2] laid out the basic analogies between food foraging and information seeking: predator (human in need of information) forages for prey (the information itself) along patches of resources and decides on a diet (what information to consume and what to ignore) based on profitability and prevalence. Pirolli raised foraging theory from biological and physical levels to knowledge and rational levels. In particular, he built the *task environment* into the underpinnings of information foraging theory. The task environment is concerned with adaptive-level analysis and models information forager's intentional constructs like goals and perceptions [2]. Pirolli has successfully applied the core mathematics of optimal foraging theory to study human behaviors during information-intensive tasks such as Web navigation [2]. As a result, information foraging theory has become extremely useful as a practical tool for Web site design and evaluation [8, 9].

Code navigation is central to software maintenance, but its support in contemporary programming environments is far from satisfactory. For example, Ko *et al.* [3] reported Eclipse's substantial navigational overhead by observing that the developers spent 35% of their time on software maintenance tasks simply navigating through the code and that only half of the searches returned task-relevant code. Research on improving code navigation has a substantial history, which can only be briefly mentioned here.

Historical analysis relies on project memory. For example, Team Tracks [1] supports collaborative filtering by recording fellow team members' navigation paths. *Static analysis* exploits the lexical, syntactic, and structural information in the source code. For example, Hipikat [10] employs a textual similarity matcher that assigns weights to words based on global prevalence of the word in the repository and local prevalence of the word in the artifact. *Dynamic analysis* leverages runtime information. For example, WhyLine [11] facilitates navigation by mapping developers' questions about a program's output to its executions.

As will be discussed in Section 3, applying information foraging theory to these tools offers a unified account for *why* the tools work. Lawrance *et al.* recently pioneered the application of information foraging to debugging [4-6]. They viewed developer as predator and bug-fix as prey. They used a well-known word similarity measure (tf-idf) between the bug report and the source code as an approximation of information scent, and demonstrated the developers' scent-following behavior by matching navigation recommendations computed by tf-idf with those actually observed. However, the theoretical underpinnings of Lawrance's model are incomplete (e.g., task environment is not mentioned) and inconsistent (e.g., their notion of a developer's evolving goals during foraging [6] contradicts the assumption of a steady goal in information foraging). Next, we tease out a more complete, but succinct, set of constructs of a unified code foraging theory.

3. A UNIFIED THEORY

Figure 2 shows our unified code navigation model in Bachman notation, a variant of an entity-relationship diagram [12]. Boxes, arrows, and ovals represent entities, relationships, and attributes respectively. The three entities are interconnected, indicating that the task and the information environments will re-shape the developer's code navigation behavior. The two attributes associated with each entity form the model's core set of concepts.

The information environment is the navigation medium with resources distributed in patches. Because there are no neutral representations available to solving a problem [13], cues, such as code comments and bug reports, are signposts that either degrade or support a developer's performance. The task environment, which has not been made explicit in previous work, assumes that code navigation fulfills software maintenance goals, such as bug fixing and refactoring. Developers will iteratively decompose a high-level goal into sub-goals and tasks, and will form hypotheses of where to go next. The task environment guides the actual navigation, and the information collected along the way will confirm or refute the hypotheses, and further refine the tasks and goals. During foraging, developers follow an information scent to reach productive patches of code. Scent might be conveyed by finding a number of matches for a search term in a package [5]. Note that the three labeled relationships in Figure 2 (informs, guides, and enriches) exemplify the situation in which the developers' behavior and their environments will *co-evolve*, each shaping the other in important ways.

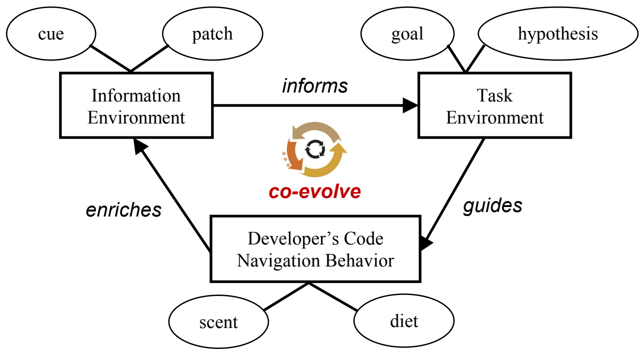


Figure 2. An information-foraging theoretic code navigation model

- Because intentional constructs like hypotheses exist only in the developer's mind, accessing and applying them in practice are difficult. The information encountered along the navigation trails often *informs* the developer's goals.
- Because developers usually have firm maintenance goals, the task environment *guides* their behaviors. This allows a steady foraging goal to direct code navigation.
- A key difference between food-foraging and information-foraging is that foragers can actively *enrich* the environment to increase the information-gain rate. For instance, developer often deliberately rearranges the IDE's layout. Thus, the profitability and prevalence of patches can be manipulated.

It is worth emphasizing that the entities and the relationships presented in Figure 2 reflect the information foraging theory's parsimony. The small set of intuitive constructs contributes to a *unified* account for code navigation. Referring back to the navigation tools reviewed in Section 2, the concept of "patches" could explain why developers collectively tend to visit files in clusters, a clue that Team Tracks relies upon. The theory also suggests why "scent", as per Hipikat's textual similarity, could be a navigation predictor. Finally, an information-foraging theoretic explanation of WhyLine's success may be that re-creating the bug's failure is such a critical debugging task that distilling the runtime semantics makes it easier to form a "hypothesis".

Although the succinct set of primitives is desired for tool builders to understand and leverage the theoretical principles, the challenge is to determine whether these principles apply to code navigation in the first place. The following list provides a minimal set of assumptions that must be examined, yet the current literature is deficient in addressing these fundamental issues.

- The task model: Is navigating code a goal-oriented foraging task?

  The points in favor include that developers who made a plan to attain maintenance goals and stuck to the plan were more successful [14]. The points against include developers' evolving goals in performing debugging tasks [6].

- The patch model: Are code fragments distributed in patches that exhibit topical localities?

  The points in favor include that source code was patched in word clusters [6]. What is not known is whether the within-patch and between-patch navigations follow foraging principles like Charnov's Theorem (cf. Section 2).

- The diet model: Do developers follow scent in finding relevant information?

  The points in favor include using word similarity to predict information scent [4]. What is not known is whether the developer's diet (what code to navigate and what to skip) conforms to the principles of profitability and prevalence [7].

The answers to these questions will provide valuable insight into information foraging theory's scope of applicability. While mapping the basic concepts (e.g., patch and scent) to code navigation requires creativity, thorough and rigorous empirical studies must be conducted to search for answers to the core software maintenance questions. This is precisely the focus of our research.

4. PRELIMINARY STUDY AND RESULTS

In a first stage of this research, we investigated the premises of the patch model. We studied SharpNLP [15], a large open-source C# project containing a collection of natural language processing (NLP) tools. The 1.0.2529 Beta release of SharpNLP under our study has 24 packages and 277 classes. We adopted two maintenance tasks from SharpNLP's issue tracker [15]. SIZE (issue #1899970) is a perfective maintenance task that requires refactoring a fixed, hard-coded prefix and suffix size to a new form of instance variables. TYPE (issue #2750882) is a corrective maintenance task that aims to fix the bug of not being able to successfully generate tags for a proper noun.

We recruited 15 developers that included both graduate students and staff programmers. All developers had previous experience with C# and were quite familiar with the application domain of SharpNLP. We asked the developers to navigate SharpNLP's code space to identify relevant code fragments that would fulfill the given maintenance tasks. The developers used only our tool for code navigation in the controlled experiment. The tool logged fine-grained user interactions, and provided two basic navigation facilities: searching by keywords and viewing (drilling-down) one particular portion of the code. Each developer worked on two maintenance tasks, and was given 5 minutes to perform each task. To control for learning effects, the order of presentation of the two tasks was counterbalanced.

Figure 3 shows the results in code navigation graphs. Due to space constraints, only two developers' navigations are reported here, chosen arbitrarily. The developer is analyzed as working in a problem space, which is defined by a set of states, a set of operators for moving between states, an initial state, a goal state, and a current state. Two problem spaces appear in our analysis.

- The "keyword" problem space's states are all search strings and search results. Moves consist of matching case, matching whole word, Boolean operator, wild card, and regular expression.
- The "view code" problem space's states are all classes. Moves consist of hitting the back button and clicking on a snippet that shows class name, number of matches, and package name.

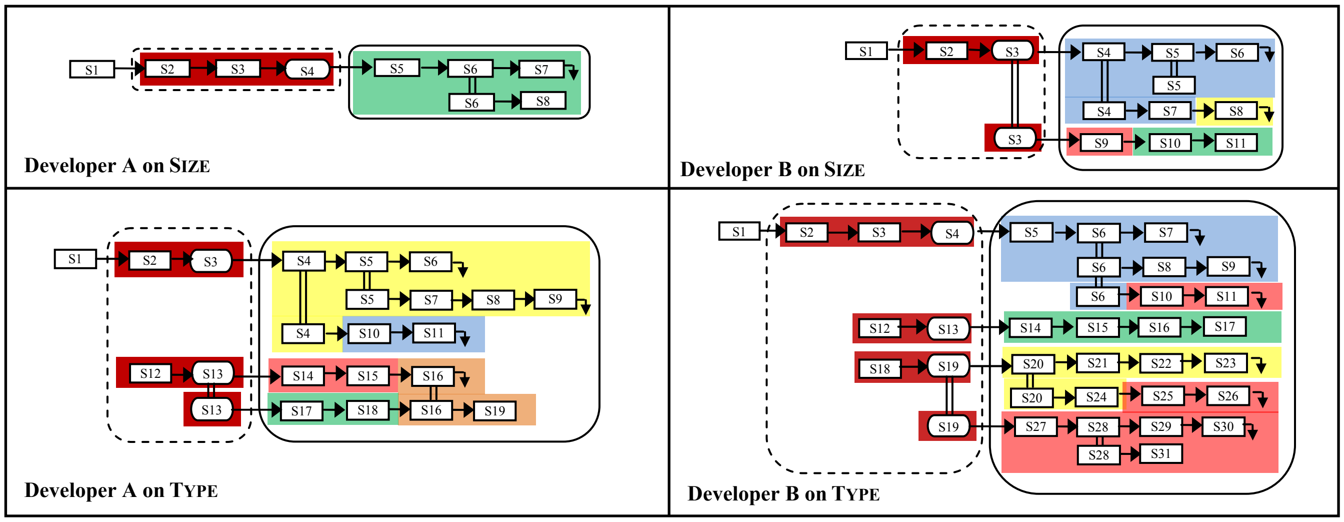


Figure 3. Code navigation graphs. Boxes are states (oval boxes are distinguished to show search result pages). Arrows are moves. Double vertical arrows are returns to a previous state. Color surrounding the boxes represents distinct packages (patches). Dotted enclosing lines show that the states and moves are part of the "keyword" problem space. Solid lines enclose the "view code" problem space.

A code navigation graph is a variant of a Web behavior graph [2]. Time in the graph proceeds left to right and top to bottom. The graph is particularly good at showing the *structure* of navigation. Similar to [5], we treat each package (i.e., a group of classes) as an information patch. Thus, each surrounding color in Figure 3 represents a distinct patch. It can readily be seen from Figure 3 that TYPE is a more difficult task than SIZE, since each TYPE task requires more states and more backtracking than the corresponding SIZE task for the same developer. Among the 24 patches, the 15 participating developers visit an average of 5.1 patches for TYPE and 2.8 patches for SIZE. This concentrated navigation pattern implies that patches indeed exhibit topical localities. Another key finding is that there are more transitions (navigations) within a patch than between patches. The ratio of within to between transitions is 3.2 for TYPE and 5.5 for SIZE. Our qualitative results therefore support the patch model's premises.

5. EMERGING RESEARCH DIRECTIONS

The need to ease program comprehension and reduce the cost of code navigation has led us to propose a novel and unified information foraging model. We traced the model's root to food foraging and conducted a preliminary study to test the model's underlying tenets. As with any research at this stage, there is much left to do. First, in-depth empirical studies are in order. Also of interest would be uncovering (un)successful trails in the code navigation graph to better visualize and analyze the search behavior. Finally, the notion of maximizing cumulative reward in reinforcement learning may contribute to a better characterization of code navigation strategies, in that the assumption of forager's "global" knowing of the environment is relaxed.

The application of optimal foraging theory hinges largely on mapping the theory's constructs to the application domain. A small number of constructs may improve the theory's parsimony, but at the expense of explanatory power or scope [5]. "Patch" in source code, for example, has already been instantiated at the class [4], package (group of classes) [5], and method (member of a class) [6] levels. As with Pirolli's work, we also expect developers to make practical use of our code foraging theory to design and evaluate navigation tools. The theoretical challenges (e.g., choosing the right constructs) and the practical ones (e.g., using proximal cues to inform navigation goals) are likely to be met only if software engineering researchers and practitioners build on each others' foundations in a principled manner.

6. REFERENCES

[1] M. Cherubini, G. Venolia and R. DeLine. Building an ecologically-valid large-scale diagram to help developers stay oriented in their code. In *VL/HCC*, pages 157-162, 2007.

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

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

[4] J. Lawrance, R. Bellamy and M. Burnett. Scents in programs: Does information foraging theory apply to program maintenance? In *VL/HCC*, pages 15-22, 2007.

[5] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy and K. Rector. How people debug, revisited: an information foraging theory perspective, *IBM Technical Report*, RC24783 (W0904-064), April 2009.

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

[7] D.Stephens and J.Krebs. *Foraging Theory*. PrincetonUniv. Press,1986.

[8] J. Nielsen. Why Google makes people leave your site faster. http://www.usiet.com/alertbox/20030630.html, June 2003.

[9] J. Spool *et al*. Designing for the scent of information. *UI Eng.*, 2004.

[10] D. Cubranic and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE*, pages 408-418, 2003.

[11] A. Ko and B. Myers. Finding causes of program output with the Java WhyLine. In *CHI*, pages 1569-1578, 2009.

[12] Wikipedia. http://en.wikipedia.org/wiki/Entity-relationship_model.

[13] N. Leveson. Intent specifications: an approach to building human-centered specifications. *TSE*, 26(1): 15-35, 2000.

[14] M. Robillard *et al*. How effective developers investigate source code: an exploratory study. *TSE*, 30(12): 889-903, 2004.

[15] SharpNLP website: http://sharpnlp.codeplex.com.