A Clustering-Based Approach to Enriching Code Foraging Environment

Nan Niu, *Senior Member, IEEE,* Xiaoyu Jin, Zhendong Niu, Jing-Ru C. Cheng, Ling Li, and Mikhail Yu Kataev

***Abstract*—Developers often spend valuable time navigating and seeking relevant code in software maintenance. Currently, there is a lack of theoretical foundations to guide tool design and evaluation to best shape the code base to developers. This paper contributes a unified code navigation theory in light of the optimal food-foraging principles. We further develop a novel framework for automatically assessing the foraging mechanisms in the context of program investigation. We use the framework to examine to what extent the clustering of software entities affects code foraging. Our quantitative analysis of long-lived open-source projects suggests that clustering enriches the software environment and improves foraging efficiency. Our qualitative inquiry reveals concrete insights into real developer's behavior. Our research opens the avenue toward building a new set of ecologically valid code navigation tools.**

***Index Terms*—Code navigation, cybernetic enrichment, foraging theory, information seeking, program investigation, software clustering.**

I. Introduction

Seeking relevant code to fulfill software maintenance tasks has become difficult and time-consuming, especially in a cybernetic environment where information and knowledge are distributed and decentralized. Studies show that even experienced developers have difficulty navigating around programs of very modest size [1]–[3]. The problem associated with code navigation is presumably worse when less experienced developers try to correct, enhance, or refactor unfamiliar code, a common situation in large and long-term software projects where team membership and responsibilities change frequently [4].

Developers typically use an integrated development environment (IDE) to investigate source code. Although modern IDEs provide some code navigation facilities, the support is far from satisfactory. For example, Eclipse's navigational tools caused significant overhead: developers spent, on average, 35% of their time on software maintenance tasks simply navigating through the code [2]. In another study, 76% of all code navigations using Eclipse referred back to locations that had already been visited, and most repeated navigations were arguably wasted interactions [5]. Simply put, using current tool support often causes the developers to become distracted, disoriented, and altogether lost during code navigation [4].

Code navigation can be supported in a wide variety of ways, ranging from basic textual queries and cross-reference searches (e.g., for all the callers of a method) to advanced tools that take advantage of the ever-growing quantities and types of software development data. Examples of advanced techniques include: 1) historical analysis (e.g., searchable project memory [4] and association rule mining of version histories [6]); 2) static analysis (e.g., semantic retrieval [7] and topology analysis of structural dependencies [8], [9]); and 3) dynamic analysis (e.g., execution slices [10] and dynamic configuration [11]). The rich collection of code navigation tools is not surprising when we consider the developers' diverse information needs that must be answered during software change tasks [12]. In fact, many approaches have leveraged more than one data source to maximize the efficiency of developers in different program investigation situations [13]. For instance, Mylyn [14] models a task context by monitoring a developer's activity (historical analysis) and extracting the structural relationships of program artifacts (static analysis).

While the development of code navigation tools has a substantial history, there has been surprisingly little work on gaining fundamental understandings of the factors affecting code navigation. The lack of a theoretical foundation has led to isolated and fragmented views of the field, and sometimes even divergent results. For example, developers who made a plan to attain maintenance goals and stuck to the plan were observed to be more successful [15], but none of the developers (regardless of success) recorded any plan or hypothesis in another program investigation study [3]. This shows serious limitation of descriptive models derived from specific observations. Thus, there is a critical need for the software engineering community to gain a unified and coherent understanding of the fundamental mechanisms that underlie the developers' code navigation behavior.

One theory that attracts much attention lately [2], [16]–[19] is information foraging theory, which uses our animal

Manuscript received December 29, 2014; revised March 19, 2015; accepted March 27, 2015. Date of publication April 22, 2015; date of current version August 16, 2016. This work was supported in part by the U.S. National Science Foundation under Grant CCF-1238336 and Grant CCF-1350487, and in part by the National Natural Science Foundation of China under Grant 61375053. This paper was recommended by Associate Editor L. D. Xu.

N. Niu and X. Jin are with the Department of Electrical Engineering and Computing Systems, University of Cincinnati, Cincinnati, OH 45221 USA (e-mail: nan.niu@uc.edu).

Z. Niu is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China.

J.-R. C. Cheng is with Information Technology Laboratory, U.S. Army Engineer Research and Development Center, Vicksburg, MS 39180 USA.

L. Li is with the Department of Information Technology and Decision Sciences, Old Dominion University, Norfolk, VA 23529 USA.

M. Y. Kataev is with the Department of Control Systems, Tomsk State University of Control Systems and Radioelectronics, Tomsk 634050, Russia.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCYB.2015.2419811

ancestors' "built-in" food-foraging mechanisms [20] to understand human information seeking and gathering in the vastness of the Web [21]. Foraging theory stems from the assumption that people (indeed, all organisms) are ecologically rational and adapt to the environments in which they operate [20], [21], and thus has great potential of offering a unified account for developers' seeking relevant code during software maintenance. In an earlier vision paper, we related foraging theory's tenets to code navigation and described some preliminary results that illuminate how the optimal foraging mechanisms appear to work as developers navigate toward their maintenance goals [22]. This paper is among the biologically inspired approaches to tackling cybernetic challenges [23]–[25].

In this paper, we present an in-depth study to investigate enrichment [21], a core mechanism for increasing foraging efficiency through manipulating resources in the environment. Our aim is to assess to what extent software (the information environment) can be rearranged to facilitate developers' (foragers') navigation. To answer the research question, we focus on software clustering by which enrichment can be done automatically. This paper tests the foraging principles in two ways. First, we perform a quantitative analysis to determine how an optimal forager's code navigation is affected by the organization (clustering) of software. In a second phase, we conduct a detailed qualitative analysis of the interaction traces recorded during programming sessions to gain insights into the real developer's behavior.

The novelties of this paper lie in the development of a framework for automatically assessing the optimal foraging principles in the context of cybernetic program investigation. The framework provides not only a theoretical foundation for understanding developers' information seeking in light of the adaptiveness of human behavior, but also a practical means of comparing and evaluating code navigation tools. In what follows, we present background information on foraging theory and software clustering in cybernetics (Section II). We then detail our research methodology in Section III. Sections IV and V describe the quantitative and qualitative evaluations, respectively. The implications of this paper are discussed in Section VI, and finally, Section VII concludes this paper.

II. Background and Related Work

A. Optimal Foraging Theory

Animals adapt, among other reasons, to increase their rate of energy intake. To do this they evolve different methods: a wolf hunts for prey, but a spider builds a web and allows the prey to come to it. Optimal foraging theory is developed in biology for analyzing the adaptive value of food-foraging strategies [20]. Optimality here refers to the strategy that maximizes the gain per unit cost. Central to optimal food foraging are the *patch model* and the *diet model*.

The *patch model* deals with predictions of the amount of time an organism would forage within a patch before leaving for another patch. Fig. 1(a) illustrates the model by presenting a hypothetical bird foraging in an environment that consists

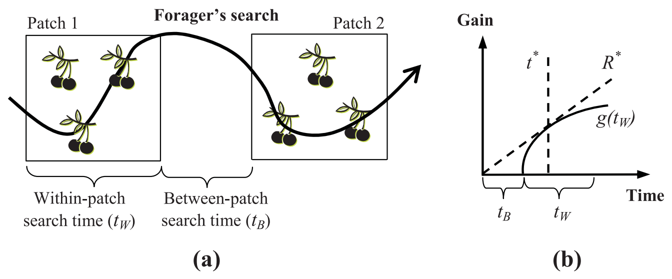


Fig. 1. (a) Illustration of patchy environment, where a hypothetical bird forages in patches containing berry clusters. (b) Charnov's marginal value theorem [20] states that the rate-maximizing time to spend in patch, t^*, occurs when the slope of the within-patch gain function $g(t_W)$ is equal to the average rate of gain, which is the slope of the tangent line R^*.

of patches of berry clusters. The forager must expend some amount of between-patch time (t_B) arriving at 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. Fig. 1(b) shows Charnov's marginal value theorem [20], which mathematically models an optimal forager's time allocation. In Fig. 1(b), $g(t_W)$ 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 rate-maximizing time, 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 follows two principles. 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.

In a nutshell, the simple rule in optimal foraging theory is: "do not expend more energy finding the food than the food provides." Animals (including humans) have evolved some very sophisticated and fascinating food-seeking mechanisms. Optimal foraging theory has been proven to be productive and resilient in addressing food-searching behavior in the field and the laboratory, whereby the adequacy of the tenets (e.g., the patch model and the diet model) is tested to account for the evolution of given structures or behavioral traits [20]. Therefore, optimal foraging theory has effectively unified many isolated studies that would otherwise not be linked in a meaningful way [20].

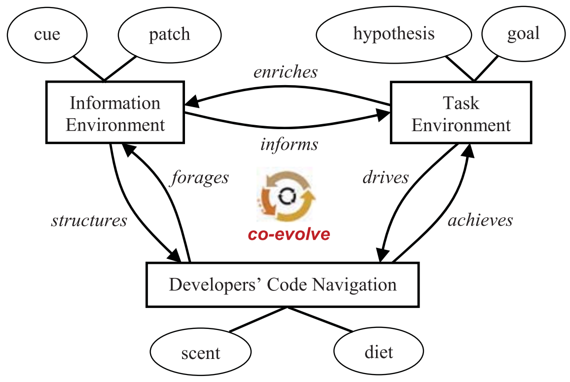


Fig. 2. Foraging-theoretic code navigation model depicted in Bachman notation, a variant entity-relationship diagram [28]. Boxes, arrows, and ovals represent entities, relationships, and attributes, respectively. This model represents a substantial refinement of our earlier work [22].

B. Foraging Theory Applied to Web and Code Navigation

Humans seeking information adopt various strategies, sometimes with striking parallels to those of animal foragers. The wolf-prey strategy bears some resemblance to classic information retrieval [26], and the spider-web strategy is like information filtering [1]. Pirolli [21] developed information foraging theory by laying out the basic analogies between food foraging and information seeking: predator (human in need of information) forages for prey (the useful information) along patches of resources and decides rationally on a diet (what information to consume and what to ignore). The theory assumes that humans are well adapted to the excessive information in the world around them, and that they have evolved strategies to efficiently find information relevant to their needs. Pirolli [21] has successfully applied the core mathematics of optimal foraging theory to study human behavior during information-intensive tasks like Web navigation. As a result, information foraging theory has become extremely useful as a practical tool for website design and evaluation [27].

Recently, information foraging theory has been applied to code navigation. Ko *et al.* [2] were among the first to relate foraging theory to developers' seeking relevant code in software maintenance; however, their study was exploratory and speculative. Lawrance *et al.* [16]–[18] recently pioneered the application of information foraging theory to debugging. They viewed developer as predator and bug-fix as prey, and presented encouraging results that matched theory's predictions with real developers' navigations. Building on these efforts, we teased out a set of constructs and assumptions of a code foraging model [22], as delineated in Fig. 2.

Fig. 2 shows that developers who investigate source code operate in two environments. The task environment is coupled with a goal, problem, or task that drives human behavior, whereas the information environment structures developers' interactions with the content. A forager's navigation is calculated according to *information scent*, a sense of value and cost of accessing a resource (e.g., an information item or a patch of items) based on perceptual cues [21]. For instance, information scent during debugging can be computed by combining: 1) a spreading activation over the link topology [8] that represents human goal memory in the task environment and 2) an interword correlation (e.g., tf-idf [26] between bug report and source code) used to approximate developer's conception of word synonymy in the information environment [16].

The succinct set of intuitive constructs presented in Fig. 2 reflects foraging theory's parsimony, which further contributes to a unified account for code navigation. For example, the concept of "patches" could explain why developers collectively tend to visit files in clusters, a clue that historical analysis tools (see [4]) rely upon. The theory also suggests why "scent," as per textual similarity in static analysis tools like [7], could be a navigation predictor. Finally, a foraging-theoretic explanation of the success of such dynamic analysis tools as [10] may be that distilling the runtime semantics makes it easier to form a "hypothesis."

In summary, foraging theory has the merit of unifying a great variety of code navigation phenomena [22]. A prominent feature of the model in Fig. 2 is that developers' behavior and their environments co-evolve, each shaping the other in important ways. While developers' adaptation to the flux of information has recently been explored [18], little is known about how the information environment can be best shaped to developers. This paper addresses the gap by investigating source code clustering, an automated enrichment method through which the information resources are rearranged to potentially improve foraging efficiency.

C. Software Clustering

Many researchers have attempted to automatically organize a large software system's structure into smaller, more manageable subsystems, giving rise to the research area of software clustering.[1] Clustering approaches have also played important roles in different aspects of cybernetics [29]–[34]. Specifically related to this paper are source code clustering techniques that facilitate program comprehension and identification of locations related to a software change. Most of these techniques employ hierarchical clustering in which few arbitrary decisions are involved [35]. Fig. 3 shows the basic steps of hierarchical clustering. Table I lists how the update rule (step 3b in Fig. 3) is defined in some well-studied algorithms [35].

Maqbool and Babri [35] reviewed the most commonly used hierarchical clustering approaches in the context of software subsystem recovery and modularization. They show that the Jaccard coefficient is one of the best similarity measures (step 2 in Fig. 3) for software clustering. They also point out that there is unlikely to be a clear winner among the many different clustering algorithms, e.g., the complete linkage (CL) algorithm produces more cohesive clusters than the single linkage (SL) algorithm, but the stability of CL is worse than that of SL. Such complementarity is viewed as a strength of clustering in that alternative views of a software system can be automatically generated [35].

Earlier work by Tzerpos and Holt [36] realized the primary goal of software clustering should be helping developers to understand the software system, rather than

[1]Approaches that cluster the entire software are described here; those clustering only the system's evolving parts are discussed in Section IV-A.

```
1. Construct an $n \times m$ entity-feature matrix ($\mathcal{M}$) to associate each
   entity $E_i$ ($1 \leq i \leq n$) with a feature vector $\mathbf{f_i} = (f_{i1}, f_{i2}, \ldots, f_{im})$.
2. Calculate similarity between a pair of entities within $\mathcal{M}$.
3. Repeat
    a. Merge the two most similar entities $E_x$ and $E_y$ into $E_{xy}$.
    b. (update rule) Treat the newly formed cluster $E_{xy}$ as a new
       entity and update the entity-feature matrix by either obtaining
       the feature vector $\mathbf{f_{xy}}$ or re-calculating similarity between
       $E_{xy}$ and all other entities in $\mathcal{M}$; this results in an updated
       entity-feature matrix of size $(n-1) \times m$.
   Until the exit criteria are met OR only one cluster remains.
```

Fig. 3. Hierarchical clustering steps.

TABLE I
UPDATE RULE DEFINED IN WELL-STUDIED ALGORITHMS

Algorithm	Update Rule
Single Linkage (SL)	sim(E_{xy}, E_z) = Max(sim(E_x, E_z), sim(E_y, E_z))
Complete Linkage (CL)	sim(E_{xy}, E_z) = Min(sim(E_x, E_z), sim(E_y, E_z))
Weighted Combined (WC)	$\mathbf{f_{xy}} = (\mathbf{f_x} + \mathbf{f_y}) / (n_x + n_y)$, where n_x and n_y denote the number of entities in E_x and E_y respectively

maximizing the value of some metric like high-cohesion or low-coupling. They developed algorithm for comprehension-driven clustering (ACDC), based on the patterns (i.e., familiar subsystem structures) frequently appeared in manual decompositions of large-scale software systems. A key contribution is bounded cardinality that ensures a reasonable cluster size to ease comprehension; thus, each resulting cluster of ACDC contains between 5 and 20 entities [36].

More recently, Scanniello and Marcus [37] presented clustering support for finding locations in source code where changes are needed to address a modification request. Structural dependencies (e.g., direct calls) are used to cluster software entities, and lexical similarities (e.g., tf-idf between change request and source code) are used to rank the entities within a cluster. Such a combination of structural and textual information has also been exploited by contemporary clustering approaches, such as [38]–[42].

III. RESEARCH METHODOLOGY

The overall goal of this research is to assess "to what extent can software clustering affect developers' code foraging?" This section describes our framework for analyzing the interplay between code foragers and their software environment. We start by refining our general research goal with specific questions concerned with the cluster-patch analysis.

A. Central Hypothesis

The patch model, one of foraging theory's most basic tenets, suggests a locality in which within-patch distances are smaller than between-patch distances [21]. This establishes a natural correspondence between a "patch" of resources (e.g., information items) and a "cluster" of data objects (e.g., software entities). Researchers have supported the patch model's premise by showing that developers' navigation was concentrated in only a small fraction of source code

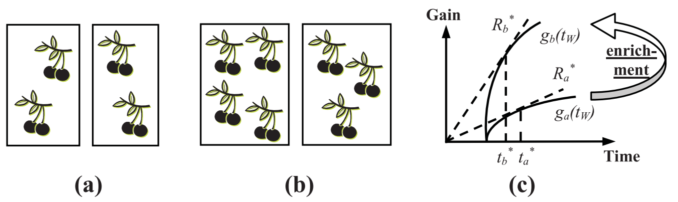


Fig. 4. In comparison to some baseline patchy environment (a), environment (b) is richer because the patches themselves are more profitable. Such an enrichment effect is illustrated in (c) in light of Charnov's marginal value theorem [20]. After enrichment, an optimal forager can leave a patch earlier ($t_b^* < t_a^*$) with more gain ($g_b(t_W) > g_a(t_W)$), i.e., can achieve an increased rate of gain.

patches [16] and that within-patch navigations were easier (and thus followed more) than between-patch navigations [22]. All the studies so far have mapped an information patch to an existing organization of source code, though at different levels: class [16], package (group of classes) [18], [22], and method (member of a class) [17]. While this creates flexibility, it is surprising to note that no current work has yet leveraged clustering methods to shape software to developers (code navigators) for a possibly more favorable arrangement of the information environment.

Our *central hypothesis* is that the way software entities are grouped (clustered) can affect the profitability of information patches, which in turn can shape the way developers navigate the code base. Fig. 4 illustrates the hypothesis that clustering more relevant entities in a patch increases foraging efficiency. From an information retrieval perspective, mean average precision (MAP) [26] can estimate patch profitability, but it provides only a static view of the information environment. To complement this, we use measures that are more directly related to developers' searching behavior. Next, we introduce such measures based on the rational analysis of optimal information foraging.

B. Rational Analysis

Anderson's rational analysis [43] is built upon the principle of rationality, which assumes that human behavior is optimally adapted to the structure and dynamics of the environment. Applied to code navigation, the principle implies that optimal developers will make the best possible navigational choices, given the information the environment makes available to them at each moment [18].

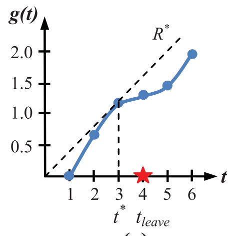
Foraging in a cluster-patch can be characterized by a cumulative gain function $g(t)$ that indicates how much information value is acquired over time t. In this paper, the information value is defined by relevant code. The proportion of relevant code (software entities) in a cluster is the precision of that cluster: $P = N_R/N_T$, where N_R is the number of relevant entities and N_T is the total number of entities in the cluster. The rate of encounter with relevant code while scanning through a list is: $\lambda_P = P/t_s$, where t_s is the time it takes to scan and judge the relevance of an information item (software entity). Let T_s and T_h be the total time spent searching and exploiting (handling relevant items), respectively. The total number of items encountered while searching is: $\lambda_P \cdot T_s$. The time

cluster
item$_1$ ✗
item$_2$ √
item$_3$ √
item$_4$ ✗
item$_5$ ✗
item$_6$ √

(a)

t	$g(t)$	slope of R
1	0	0
2	0.67	0.33
3	1.2	**0.4**
4	1.33	0.33
5	1.43	0.29
6	2	0.33

(b)



(c)

Fig. 5. (a) Cluster-patch containing six ranked information items: three relevant (marked with "√") and three irrelevant ("✗"). (b) Cumulative gain, $g(t)$, obtained at each time step, $t \in [1, 6]$, when foraging in patch (a): $t_B = t_s = t_h = 1$ (time unit); slope of R, the line passing through the origin (0, 0) and (t, $g(t)$), indicates the marginal value. (c) According to Charnov's theorem [20], the maximal rate of gain is achieved at the tangency point: $t^* = 3$, and an optimal forager will leave patch (a) at $t_{\text{leave}} = 4$ when a decrease of the derivative (instantaneous slope) of the gain function occurs.

to handle a relevant item is: $t_h = T_h/(\lambda_P \cdot T_s)$. Considering these factors in the calculation of the cumulative gain function, we have

$$\begin{aligned} g(t) &= t \cdot \left(\frac{\lambda_P \cdot T_s}{T_s + T_h}\right) = t \cdot \left(\frac{\lambda_P \cdot T_s}{T_s + t_h \cdot \lambda_P \cdot T_s}\right) \\ &= t \cdot \left(\frac{\lambda_P}{1 + t_h \cdot \lambda_P}\right) = t \cdot \left(\frac{\frac{N_R}{N_T \cdot t_s}}{1 + t_h \cdot \frac{N_R}{N_T \cdot t_s}}\right) \\ &= \frac{N_R \cdot t}{N_T \cdot t_s + N_R \cdot t_h}. \end{aligned} \tag{1}$$

Fig. 5(a) illustrates the rational analysis with a patch of six ranked items. We assume the cost of between-patch search, t_B (see Fig. 1), and within-patch scanning (t_s) and handling (t_h) to be one time unit, i.e., $t_B = t_s = t_h = 1$.[2] Following (1), Fig. 5(b) details forager's cumulative gain at each time step. The gain function is further plotted in Fig. 5(c) where the optimal time to leave the patch is marked with ★. Formally, $t_{\text{leave}} = \lfloor t^* \rfloor + 1$, where t^* is the rate-maximizing time defined in Charnov's marginal value theorem [20]. We use two measures to assess foraging efficiency.

1) *Missed Diet:* Proportion of relevant items not navigated to the total number of relevant items.
2) *Navigational Overhead:* Proportion of navigated items which are irrelevant.

For the example shown in Fig. 5, the set of all relevant items is: $\{\text{item}_2, \text{item}_3, \text{item}_6\}$, and the items that an optimal forager navigates are: $\{\text{item}_1, \text{item}_2, \text{item}_3, \text{item}_4\}$. Therefore, the missed diet is: $(|\{\text{item}_6\}|)/(|\{\text{item}_2, \text{item}_3, \text{item}_6\}|) = 1/3 = 33\%$, and the navigational overhead is: $(|\{\text{item}_1, \text{item}_4\}|)/(|\{\text{item}_1, \text{item}_2, \text{item}_3, \text{item}_4\}|) = 2/4 = 50\%$.

IV. Quantitative Analysis

The objective of our quantitative analysis is to rigorously assess software clustering's impact on code navigation.

A. Relevance Determination

Relevance in this paper [as illustrated by "√" in Fig. 5(a)] refers to source code that should be navigated together during a software change task. We devise a surrogate measure of relevance by mining the co-change relationship over a software project's revision history. The intuition is that software entities frequently and consistently changed together in the past are worth considering (navigating) jointly for addressing future modification requests.

Co-changed entities can be identified by various data mining techniques, e.g., association rule mining [6]. Among the clustering-based approaches, Beyer and Noack [44] focused on change localization so that different clusters can evolve independently, Hassan and Holt [45] used the co-change information to ensure software changes are correctly propagated, and Robillard and Dagenais [46] clustered the change sets to support program investigation. These approaches all require a large history of changes as input, and the output involves only the evolving parts of the system.

To automatically determine relevance, we adapt CCVisu, a tool that reads software change history and computes clusters of co-changed files [44]. Empirical evaluation shows CCVisu's clustering results conform largely with the authoritative decomposition prepared manually by experts. This makes CCVisu a reliable source of evolutionary clusters [47] when expert opinions are unavailable. In devising the relevance measure, our overarching goal is accuracy, i.e., we want the obtained co-change relationship to be as accurate as possible. To that end, we apply the following steps.

1) Skip the first 200 change sets, a strategy used in [45] and [46] to discard unstabilized modification records.
2) Run CCVisu on only the successfully closed modification records to eliminate noisy input's effect.
3) Keep only the "most frequently changed"[3] files in CCVisu's results because the ones changed less often can sometimes be clustered incorrectly [44].
4) Remove the files that do not appear in the version being clustered and make sure that the resulting clusters conform with known authoritative views.

B. Project Selection

Systems are selected among the long-lived projects with extensive change histories. Having authoritative views of the system decomposition is only an optional inclusion criterion. We use theoretical sampling [48] to achieve diversity and comparability. Diversity is incorporated along such dimensions as problem domain, programming language, and change repository. Comparability is facilitated by considering systems of the same family. Table II shows the selected projects: Ant and Ivy are from the Apache family, Firefox is written primarily in C/C++ and JavaScript, and Mylyn uses an online issue tracking system (Bugzilla). We perform clustering only to software's latest release. Table II lists the number of source code files to be clustered for each project.

Table II also reports the projects' characteristics related to our CCVisu analysis. The "successfully closed change sets" column gives CCVisu's input. The output uncovers the co-change relationship among the "changed files." To ensure accuracy of such relationship, we employ two heuristics to

[2]Section V sheds light on parameter values from the real developer.

[3]Section IV-B presents heuristics for the most frequently changed files.

TABLE II
CHARACTERISTICS OF SELECTED PROJECTS

Project	Domain	Source	Prog Lang	Latest release version	# of code files	Software change repository	# of successfully closed change sets (time span)	# of changed files	# of most frequently changed files	# of CCVisu clusters
Ant	Software build	ant.apache.org	Java	1.8.2	1276	Subversion	12366 (07/2000-07/2011)	828	129	5
Ivy	Relation mgmt	ant.apache.org/ivy	Java	2.2.0	626	Subversion	1874 (04/2005-07/2011)	363	73	4
Firefox	Web browsing	mozilla.org/firefox	C/JS	3.6.1	1968	CVS	14475 (01/1999-07/2011)	740	148	8
Mylyn	Task mgmt	eclipse.org/mylyn	Java	3.6.22	2321	Bugzilla	4819 (07/2007-02/2010)	1221	144	7

post-process CCVisu's output by keeping only the most frequently changed files. The first takes the form of the "80/20 rule" because we find that the distributions of change frequency of files follow a power law [49], i.e., 20% of the files are changed 80% of the time. The second heuristic is derived from our work on using entropy to determine information values [50], where we find that one standard deviation (σ) above the mean (μ) represents a threshold for significance. Thus, for the current analysis, let $\langle f_1, f_2, \ldots, f_n\rangle$ be the list of all the changed files sorted by the descending order of change frequency, i.e., $\text{Freq}(f_1) \geq \text{Freq}(f_2) \geq \cdots \geq \text{Freq}(f_n)$, we take $\langle f_1, \ldots, f_m\rangle$ to be the most frequently changed files, where $m = \min(20\% \cdot n, p)$ and $p \in [1, n]$ such that $(\text{Freq}(f_p) \geq \mu_{\text{Freq}} + \sigma_{\text{Freq}}) \wedge (\forall q \geq p, \text{Freq}(f_q) < \mu_{\text{Freq}} + \sigma_{\text{Freq}})$. The last two columns of Table II list the number of most frequently changed files, and the number of co-change clusters into which CCVisu groups the most frequently changed files.

Among the four selected projects, we are aware of a known authoritative decomposition for Firefox [51]. Consistent with our expectation, the eight CCVisu clusters accurately reflect how the 148 files are authoritatively grouped. This increases our confidence in CCVisu's reliability of clustering a project's most frequently changed files. Therefore, relevance in our analysis is determined by CCVisu clusters without further manual adjustment.

C. Experimental Setup

The independent variable in this paper is software clustering, which has five values: 1) ACDC (comprehension driven); 2) SL; 3) CL; 4) weighted combined (WC); and 5) package based (PACK). ACDC is chosen because it is one of the few clustering methods to support developers' program comprehension [36]. We adopt ACDC's implementation from [52] and enable all its patterns (body-header, subgraph-dominator, and orphan-adoption) when executing it on the projects listed in Table II.

The SL, CL, and WC are chosen because they are well-known algorithms underlying many software architecture recovery techniques [35]. We use the Jaccard similarity measure to implement these hierarchical clustering algorithms and specify the exit criteria (see Fig. 3) based on ACDC's bounded cardinality (i.e., each resulting cluster should contain [5, 20] entities) [36]. Entities are clustered at the file level since this paper deals with large systems with hundreds and thousands of source code files (see Table II). Similar to [35], formal features—inherits, calls, and references—are used to determine a similarity between software entities.

TABLE III
NUMBER OF CLUSTERS k, THE AVERAGE CLUSTER SIZE $|C|$, AND THE MoJo DISTANCE TO THE PACKAGE-BASED BASELINE CLUSTERING

Project		PACK	ACDC	SL	CL	WC
Ant	k	43	44	48	47	47
	$\|C\|$	18.9	18.6	17.0	17.4	17.5
	MoJo	–	460	569	695	568
Ivy	k	23	20	19	17	17
	$\|C\|$	15.0	17.2	18.3	19.6	19.4
	MoJo	–	220	268	284	255
Firefox	k	114	116	98	109	119
	$\|C\|$	16.8	15.3	19.9	17.6	16.0
	MoJo	–	339	547	788	551
Mylyn	k	122	131	134	115	116
	$\|C\|$	18.8	17.6	17.2	19.9	19.7
	MoJo	–	469	536	665	562

The baseline method considered in this paper is PACK, which leverages the package/directory structure to cluster source code files. In implementing PACK, our aim is to mimic the grouping that modern IDEs "package explorer" provides to the developer. While restricting [5, 20] files per cluster seems a reasonable approximation of package explorer's size, we also attempt to preserve the file hierarchy by organizing the files based on a depth-first search scheme.

Table III summarizes the clustering results. Due to the incorporation of bounded cardinality [36], all the clustering methods tend to produce relatively uniformly sized clusters. This is a nice property when considering usability of the clusters for supporting comprehension-driven tasks [53] such as code navigation. In order to compare the clustering results, we use the MoJo distance measure [54]. MoJo measures the distance between two decompositions of the same software system by computing the number of Move and Join operations to transform one to the other. Intuitively, the smaller the MoJo distance, the closer the two clustering results. Table III lists the MoJo distance to the result of PACK, the package-based baseline decomposition. The average pairwise MoJo distance is 539.16 among the five clustering methods performed on each of the four subject software systems. We believe this distance can be attributed to the different features used to determine source code files' similarity during clustering: PACK relies solely on the directory structure, ACDC employs the subsystem patterns, and the hierarchical clustering algorithms, SL, CL, and WC, exploit static dependencies in distinct manners.

Having produced software clusters, we are left with the task of ranking the entities within a cluster in order to perform the optimal foraging analysis (see Fig. 5). In an attempt to select and weight features for improving clustering results, Andritsos and Tzerpos [55] reported that the tf-idf scheme

TABLE IV
RESULTS OF SOFTWARE CLUSTERING'S EFFECT ON CODE NAVIGATION

Project		MAP	Missed Diet	Navigational Overhead	Project		MAP	Missed Diet	Navigational Overhead
Ant	PACK	$24\% \pm 18\%$	$17\% \pm 9.5\%$	$91\% \pm 12.8\%$	Ivy	PACK	$26\% \pm 13\%$	$18\% \pm 4.0\%$	$87\% \pm 11.2\%$
	ACDC	$83\% \pm 26\%$	$\boxed{10\% \pm 7.0\%}$	$61\% \pm 29.2\%$		ACDC	$77\% \pm 29\%$	$4\% \pm 3.2\%$	$\boxed{31\% \pm 19.3\%}$
	SL	$79\% \pm 25\%$	$12\% \pm 8.9\%$	$67\% \pm 21.4\%$		SL	$80\% \pm 27\%$	$9\% \pm 5.1\%$	$33\% \pm 13.6\%$
	CL	$28\% \pm 42\%$	$16\% \pm 9.3\%$	$56\% \pm 15.6\%$		CL	$72\% \pm 27\%$	$5\% \pm 2.9\%$	$38\% \pm 17.8\%$
	WC	$\boxed{84\% \pm 21\%}$	$13\% \pm 5.0\%$	$\boxed{47\% \pm 11.9\%}$		WC	$\boxed{82\% \pm 26\%}$	$\boxed{2\% \pm 0.9\%}$	$40\% \pm 12.2\%$
	ANOVA Results	$F(4, 109) = 66.9$ $p = 1.6 \cdot 10^{-39}$	$F(4, 109) = 0.2$ $p = 0.9$	$F(4, 109) = 34.6$ $p = 2.0 \cdot 10^{-23}$		ANOVA Results	$F(4, 60) = 13.7$ $p = 5.5 \cdot 10^{-8}$	$F(4, 60) = 11.5$ $p = 5.4 \cdot 10^{-7}$	$F(4, 60) = 10.4$ $p = 1.9 \cdot 10^{-6}$
Firefox	PACK	$43\% \pm 31\%$	$19\% \pm 7.0\%$	$70\% \pm 14.0\%$	Mylyn	PACK	$21\% \pm 12\%$	$19\% \pm 13.4\%$	$91\% \pm 10.4\%$
	ACDC	$61\% \pm 35\%$	$28\% \pm 7.2\%$	$\boxed{26\% \pm 20.5\%}$		ACDC	$73\% \pm 34\%$	$\boxed{14\% \pm 6.3\%}$	$62\% \pm 20.5\%$
	SL	$75\% \pm 34\%$	$11\% \pm 9.7\%$	$60\% \pm 14.4\%$		SL	$63\% \pm 32\%$	$25\% \pm 8.0\%$	$66\% \pm 12.5\%$
	CL	$\boxed{79\% \pm 31\%}$	$13\% \pm 8.4\%$	$45\% \pm 25.9\%$		CL	$62\% \pm 33\%$	$32\% \pm 8.5\%$	$67\% \pm 15.7\%$
	WC	$74\% \pm 30\%$	$\boxed{9\% \pm 2.1\%}$	$65\% \pm 14.3\%$		WC	$\boxed{75\% \pm 35\%}$	$21\% \pm 12.1\%$	$\boxed{52\% \pm 21.0\%}$
	ANOVA Results	$F(4, 273) = 2.1$ $p = 0.09$	$F(4, 273) = 0.8$ $p = 0.6$	$F(4, 273) = 7.9$ $p = 5.2 \cdot 10^{-5}$		ANOVA Results	$F(4, 276) = 33.2$ $p = 1.2 \cdot 10^{-22}$	$F(4, 276) = 2.3$ $p = 0.8$	$F(4, 276) = 16.4$ $p = 4.7 \cdot 10^{-12}$

commonly used in text retrieval [26] is best suited for software clustering. We, therefore, use tf-idf as a basis for determining the ranking inside a cluster; this is also in line with the recent work on clustering support for static concept location [37]. In particular, we apply our source code indexing tool [56] to convert each file to a vector in the corpus. We then treat each authoritative (i.e., CCVisu co-change) cluster as a query, and use the tf-idf score to rank the entities inside a cluster-patch. Note that the selected projects have multiple authoritative clusters (queries), as shown in Table II. This allows for the computation of mean average precision introduced next.

D. Results

As discussed in Section III, we use three dependent variables to assess clustering's effect on code navigation: MAP, missed diet, and navigational overhead. All three are ratio measures that range from 0 to 1. In this paper, each CCVisu co-change (co-navigation) relationship represents a unit of analysis for the software project. Table IV reports the analysis results. Descriptive statistics are given in terms of (mean ± standard deviation). The best results are highlighted by rectangular boxes.

To answer the question, "Does software clustering enrich code navigation environment?" we perform repeated measures analysis of variance (ANOVA) and planned comparison tests for the means. These tests are based on an assumption that the population is normally distributed. The law of large numbers states that if the population sample is sufficiently large (e.g., more than 50), then the central limit theorem applies even if the population is not normally distributed [49]. In this paper, even the smallest project (Ivy) offers 60 independent observations, therefore, the central limit theorem applies and the above mentioned tests have statistical significance. In Table IV, inferential statistics are given by ANOVA results. The results are displayed in dark background if they are statistically significant at $\alpha < 0.01$ level and in gray otherwise.

1) MAP estimates an information environment's overall profitability since it measures "the quality across the recall levels" [26]. Intuitively, the higher the MAP, the closer the true positives (relevant code) are to the top of the information patches (source code clusters). Table IV shows that the WC algorithm gives the highest MAP for all the projects except for Firefox. The MAP values of PACK are the lowest across all the projects. The result suggests software clustering based on more advanced features (e.g., static dependencies and subsystem patterns) than only the package/directory structure is likely to improve patch profitability of the information environment.
2) Missed diet evaluates how much relevant code an optimal forager fails to attend to. It quantifies omission errors. According to Table IV, ACDC and WC result in the minimum missed diet, but the difference is significant only for the smallest project (Ivy). It is therefore interesting to note that no matter how the software entities are clustered (rearranged), an optimal forager will likely miss similar amounts of relevant code when navigating large software projects.
3) Navigational overhead assesses an optimal forager's effort expended in investigating irrelevant code. It quantifies commission errors. Table IV reveals that the least overhead is experienced when foraging in the patches generated by WC and ACDC. ANOVA planned comparisons show that the baseline method (PACK) causes significantly more navigational overhead than the ACDC, SL, CL, and WC clustering methods for all the projects.

It is important to point out here that missed diet and navigational overhead are novel metrics that hinge on the optimal forager's behavior; to be exact, t_{leave} determined by Charnov's theorem (see Fig. 5). In another word, they are not simply the complements of recall and precision commonly used in information retrieval [26], but are defined based on Anderson's rational analysis [43]. While MAP is traditionally an information retrieval metric, its use of estimating patch profitability is new. In fact, we recently updated profitability calculation by combining precision and MAP together [19].

The results presented above are concerned with all the clusters containing relevant code. In practice, developers follow only the best code navigation choices [18]. Such selectivity, therefore, reflects the task specificity of code navigation, and is

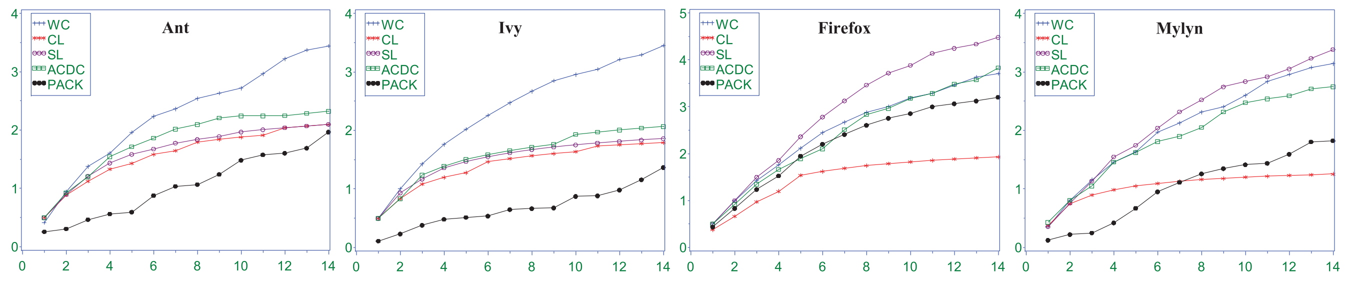


Fig. 6. Most profitable patch analysis: x-axis represents time (t) and y-axis represents the cumulative gain function ($g(t)$).

also consistent with foraging theory's prediction that unprofitable patches will be ignored by an optimal forager [20]. For this reason, Fig. 6 plots the cumulative gain over 14 time steps, the largest common cluster size of the projects' most profitable patches. The plots offer further insight into the findings. For example, PACK's nearly linear accelerating rate indicates the amount of relevant code does not diminish as foraging progresses. This causes the forager to leave PACK's patch late (i.e., large t_{leave} value)—the main reason contributing to PACK's large navigational overhead.

A practical issue here is that the oracle of determining information item's relevance is known *a priori* in our quantitative analysis. In nonexperimental settings, the profitability measures (oracles) may not be directly computed based on the external quality of an information patch. To ameliorate this situation, our recent work has shown that patch cohesion—one of the internal quality indicators—can serve as the *perceived* profitability to offer much practical value [19].

In summary, our quantitative analysis suggests that, compared to a basic organization of software entities (PACK), clustering does improve patch profitability and foraging efficiency. Although no single clustering method is the best for all the projects by all the measures, WC appears to perform consistently well when examined for all the patches (see Table IV) as well as for the most profitable ones (see Fig. 6). We recommend WC as a starting point (or a new baseline) for practitioners and tool builders interested in using clustering to enrich code navigation environment.

E. Threats to Validity

As is the case for most controlled experiments, our quantitative assessment of software clustering's impact on code navigation is performed in a restricted and synthetic context. We discuss here some of the most important factors that must be considered when interpreting the results.

The construct validity [48] of our analysis can be affected by the use of CCVisu to operationally measure "relevant code navigated together." In the absence of detailed programming interaction traces, mining a long-lived project's change repository provides a robust and objective way to evaluate code navigation approaches. It should be noted that our restrictive gauge choice based on the most frequently changed code grossly underestimates the performance of the software clustering methods, because developers may find less frequently changed or even unchanged code "relevant" for carrying out software change tasks. While this should have little effect on the comparisons, caution must be taken in interpreting the absolute values of the results.

We believe the main strength of our experimental design is its high internal validity [48]: soundness of the relationship between independent and dependent variables. Because all the factors potentially affecting both structural (MAP) and behavioral (missed diet and navigational overhead) measures are under our direct control, any significant difference must be caused by the different clustering methods employed.

The results of our analysis may not generalize to other software projects—a threat to the external validity [48]. Our chosen systems are all open-source projects due to the availability of their revision repositories. However, these are not necessarily representative of all systems and, in particular, proprietary software products are likely to exhibit different characteristics. We also note that some well-established software clustering methods (e.g., BorderFlow [37] and Bunch [57])[4] are not studied and that our studied methods cluster source code files by using only static features. Such decisions are intentional because the BorderFlow and Bunch algorithms are nondeterministic and using only static features allows incomplete or unexecutable legacy programs to be analyzed. Thus, it is not clear how the results might generalize to other software clustering approaches.

It is worth stressing at this point that we have made a few simplifying assumptions in our quantitative analysis, most notably that it takes one time unit to scan and handle a software entity. In practice, there may be considerable variation in searching time. The analysis described in the next section sheds light on the impact and the potential update of this important assumption.

V. Qualitative Analysis

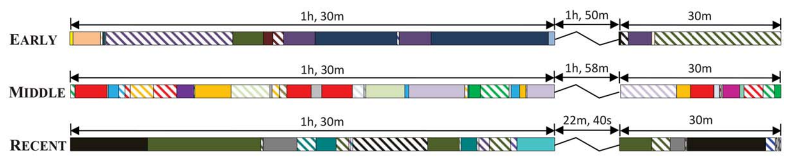
The results of Section IV provide an objective and precise assessment of the degree to which an optimal forager and the environment co-evolve. However, the behavior of real developers often departs from that of the optimal forager. We complement the quantitative analysis by conducting a qualitative study of developers' actual behavior.

The objective of our qualitative analysis is to understand the evolution of a developer's code navigation. We focus on the scanning time (t_s) and the handling time (t_h) since we have made a simplifying assumption about them in Section IV, namely, $t_s = t_h = 1$ (time unit). In our current qualitative

[4]LIMBO (scaLable InforMation BOttleneck algorithm [55], another established method, behaves the same as WC [35].

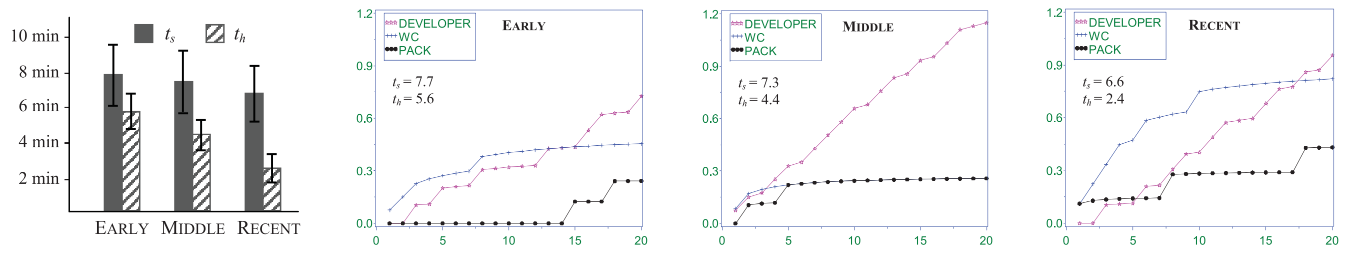
Stage	Task description	Task commit date	Mylyn version modified	# of files viewed	# of files edited	Naviga-tional overhead
EARLY	Add ETag attribute	Dec 5, 2007	2.2.0	12	9	25%
MIDDLE	Enhance strike-through font	Sep 24, 2008	3.0.0	7	6	14%
RECENT	Enable language configurations	Dec 19, 2009	3.3.1	9	9	0%

(a)



(b)

Fig. 7. (a) Selected enhancement tasks successfully completed by the same Mylyn developer at three different stages. (b) Source code files sought during the three selected programming sessions reported in (a). Every session involves a sequence of files (boxes). Time proceeds left to right. Color of the boxes represents distinct files. Box length indicates how long the file operation is. Fully shaded box means the file is viewed but not edited. Stripe-shaded box shows the file is both viewed and edited. Due to space limit, we show the developer's behavior only for the session's first 1.5 and last 0.5 h.



(a) (b) (c) (d)

Fig. 8. (a) Characterizing the t_s and t_h values. The value of the bar represents the mean and the thin vertical line at the top of the bar represents the standard error of the mean. (b)–(d) Updating the most profitable patch analysis (see Fig. 6) by applying the mean values of t_s and t_h reported in (a) to the different stages: ($t_s = 7.7$, $t_h = 5.6$) applied to EARLY, ($t_s = 7.3$, $t_h = 4.4$) applied to MIDDLE, and ($t_s = 6.6$, $t_h = 2.4$) applied to RECENT.

study, t_s and t_h refer to the amount of time a source code file is viewed and edited, respectively. We take advantage of Mylyn's detailed interaction traces of over 4000 programming sessions. We use purposive sampling [48] to select three enhancement tasks performed successfully by the same developer but at different stages of the developer's interaction with the project. We label these stages as "EARLY," "MIDDLE," and "RECENT."

Fig. 7(a) provides the basic information of the selected tasks. The developer (Frank) is a key Mylyn contributor who has completed many enhancement tasks. Although the number of edited files is comparable among the tasks, the navigational overhead decreases from EARLY to RECENT. In carrying out a RECENT task, for example, the developer views only relevant code with no navigational overhead. Fig. 7(b) details developer interaction. Note that, after inspecting some random samples of Mylyn programming sessions, we treat the interactions whose elapsed time is less than 5 s or greater than 30 min as outliers and remove them from Mylyn's raw interaction log. It is reported that a Java developer estimates that an average cycle takes 31 min [58]. Fig. 7(b) visualizations can be understood in the context of developers' editing behavior: a recent study shows that developers tend not to "edit-first," but seem to "edit-throughout," or "edit-last," when performing enhancement tasks [59].

We use a 30-day window to approximate the developer's searching time at different stages. Specifically, we collect all the successful enhancements made by Frank 15 days before and after the selected task's commit date [see Fig. 7(a)]. From this collection, we compute the average t_s and t_h values for each stage. Fig. 8(a) shows the results, where the average time spent editing a file (t_h) is reduced markedly. Surprisingly, the viewing time's decrease is not nearly at the same rate. Although preliminary, the finding illuminates that one essential complexity of code navigation lies in seeking relevant code rather than handling the actual modification. Once the relevant code is found, the productivity of making software changes can be expected to increase as the developer becomes more familiar with the system.

Identifying the real developer's searching time allows us to refine the rational analysis. Here, instead of assuming $t_s = t_h = 1$, Equation (1) can now be instantiated with Fig. 8(a) stage-specific (t_s, t_h) value pairs. In addition to this change, other updates from our quantitative analysis presented in Section IV are included in the following.

1) Relevance is determined not by CCVisu's output but by the co-changed files committed by the developer.
2) Software clustering is performed not on Mylyn's latest release but on the three versions upon which the selected tasks are performed [see Fig. 7(a)].
3) The tf-idf score is calculated not against an entire authoritative cluster's indices but against the feature request of the actual enhancement task.

Fig. 8(b)–(d) show the refined results of the most profitable patch analysis. These plots depict the developer's actual behavior, as well as the performance of the baseline (PACK) and our recommended new baseline (WC) clustering methods. The PACK and WC curves show the gains predicted by optimal foraging theory when the developer navigates through the most profitable cluster-patch generated by PACK and WC. The number of plotted time steps is 20 since it is the upper bound of cluster size in this paper.

In accordance with our quantitative analysis results, WC performs better than PACK in all stages. It is encouraging to realize the developer's EARLY code navigation is close to WC's prediction [Fig. 8(b)], demonstrating the usefulness of the best navigation choices recommended by software clustering.

It is even more encouraging to realize the developer's RECENT departure from WC's prediction [Fig. 8(d)], suggesting an improved gain is possible via the enrichment of the code navigation environment. In Fig. 8(c), the real developer promptly outperforms the theory's prediction of the optimal forager's search in both clustering environments, illustrating a situation where automated support offers limited help.

In summary, our qualitative analysis provides initial evidence that developers become adapted to their code navigation environment, and that it is possible for better assisting them to evolve the strategies to maximize the gains of relevant code per unit cost. Our qualitative inquiry is best viewed as an exploratory case study [48] of discovering the evolutionary aspects of real developer's behavior and feeding those aspects back into the rational analysis of developer's code navigation. The results are subject to theoretical rather than statistical generalizations. Nevertheless, due to our purposive sampling, much remains to be explored, e.g., what is the impact of the task type (enhancement, bug fixing, and refactoring)? The task difficulty, priority, and severity? The editing habit? The unsuccessfully completed tasks? And the interactions lasting less than 5 s or more than 30 min?

VI. IMPLICATIONS FOR THEORY AND TOOL SUPPORT

Research on code navigation has resulted in some *ad hoc* tools without any theoretical basis, along with some descriptive models derived from specific observations of developer behavior. These approaches have produced only modest improvements, and led to rather isolated efforts that tackle only a few phenomena and that have little theoretical content in common with the work of others. We believe information foraging—an evolutionary-ecological theory studying human's adaptive interaction with information—has great potential to unify the field of code navigation [22]. The appeal for our development of a unified theory (see Fig. 2) is that a single set of mechanisms can both account for all the descriptive models and lead to principled ways to increase practical tool support for software developers.

Earlier theories of programmer behavior (see [60], [61]) rely primarily on such "in-the-head" constructs as mental models to account for code navigation. Later theories (see [2], [18]) shifts from largely in-the-head program comprehension toward "in-the-environment" cognition. Rational analysis [43] holds the key to understanding programmer navigation without reference to complex mental states. The basic idea is that the constraints of the environment place important shaping limits on the rationality that is possible. This paper sheds light on such bounded rationality [62] by providing a concrete computational model for understanding the machinery underlying code navigation.

Building on foraging theory's success of explaining how people seek relevant information on the Web [21], researchers have designed practical tools to help website creators and users [27]. In a similar vein, our unified theory can guide the design and evaluation of code navigation tools. As an exemplar, this paper leverages foraging theory's mathematical model to automatically evaluate enrichment methods. This evaluation framework can not only compare software clustering methods, but also assess a broad spectrum of code rearrangement tools, such as Weta [3], Code Bubbles [5], and the like. Moreover, our empirical studies have suggested new insights into designing better navigation support. For example, to reduce the "missed diet" (i.e., relevant code developers fail to attend to), the tool can spotlight the scent (e.g., salient or task-relevant features) of a resource and increase higher profitability prey's "prevalence" ($\lambda = 1/t_B$). Finally, it is crucial for tool builders and evaluators to consider the adaptiveness of developer behavior and to revisit and even update the key assumptions (e.g., the t_s and t_h values in this paper).

VII. CONCLUSION

The main contributions of this paper are the evolutionary-ecological understanding of the fundamental mechanisms underlying developers' code navigation behavior, the development of a novel framework for automatically assessing the optimal foraging principles in the context of source code investigation, the empirical evaluation of clustering-based enrichment methods, the concrete insights of real developer's navigation, and the avenues opened up for software researchers, practitioners, and tool creators.

Information foraging is about understanding and improving the interplay of people and their information environments [21]. The application of foraging theory in software engineering has focused predominantly on enabling developers to best shape themselves to the software and task environments [16]–[18]. In this paper, we have tried to make a start at reversing the foraging-theoretic thinking in software engineering—exploiting clustering to enable the environments to be best shaped to the developers. It is hoped this line of research will give us a new set of ecologically valid tools to improve developers' code navigation.

ACKNOWLEDGMENT

The authors would like to thank G. Bradshaw, A. Mahmoud, T. Bhowmik, and S. Reddivari for comments on earlier drafts of this paper.

REFERENCES

[1] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," in *Proc. ACM Symp. Softw. Vis. (SOFTVIS)*, St. Louis, MO, USA, May 2005, pp. 183–192.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[3] K. D. Sherwood, "Path exploration during code navigation," Master's thesis, Dept. Comput. Sci., Univ. Brit. Columbia, Vancouver, BC, Canada, 2008.

[4] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput. (VL/HCC)*, Dallas, TX, USA, Sep. 2005, pp. 241–248.

[5] A. Bragdon *et al.*, "Code Bubbles: A working set-based interface for code understanding and maintenance," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, Atlanta, GA, USA, Apr. 2010, pp. 2503–2512.

[6] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[7] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *Proc. Int. Conf. Program Compre. (ICPC)*, Passau, Germany, Jun. 2012, pp. 183–192.

[8] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, Aug. 2008.

[9] N. Niu, J. Savolainen, T. Bhowmik, A. Mahmoud, and S. Reddivari, "A framework for examining topical locality in object-oriented software," in *Proc. IEEE Comput. Softw. Appl. Conf. (COMPSAC)*, Izmir, Turkey, Jul. 2012, pp. 219–224.

[10] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating program features using execution slices," in *Proc. IEEE Symp. Appl. Spec. Syst. Softw. Eng. Technol. (ASSET)*, Richardson, TX, USA, Mar. 1999, pp. 194–203.

[11] F. Tao *et al.*, "Concept, principle and application of dynamic configuration for intelligent algorithms," *IEEE Syst. J.*, vol. 8, no. 1, pp. 28–42, Mar. 2014.

[12] J. Sillito, G. C. Murphy, and K. D. Volder, "Questions programmers ask during software evolution tasks," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Portland, OR, USA, Nov. 2006, pp. 23–33.

[13] L. Ke, Q. Zhang, and R. Battiti, "Hybridization of decomposition and local search for multiobjective optimization," *IEEE Trans. Cybern.*, vol. 44, no. 10, pp. 1808–1820, Oct. 2014.

[14] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Portland, OR, USA, Nov. 2006, pp. 1–11.

[15] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 889–903, Dec. 2004.

[16] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, Florence, Italy, Apr. 2008, pp. 1323–1332.

[17] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart, "Reactive information foraging for evolving goals," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, Atlanta, GA, USA, Apr. 2010, pp. 25–34.

[18] J. Lawrance *et al.*, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 197–215, Feb. 2013.

[19] N. Niu, A. Mahmoud, Z. Chen, and G. Bradshaw, "Departures from optimality: Understanding human analyst's information foraging in assisted requirements tracing," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, May 2013, pp. 572–581.

[20] D. W. Stephens and J. R. Krebs, *Foraging Theory*. Princeton, NJ, USA: Princeton Univ. Press, 1986.

[21] P. Pirolli, *Information Foraging Theory: Adaptive Interaction With Information*. New York, NY, USA: Oxford Univ. Press, 2007.

[22] N. Niu, A. Mahmoud, and G. Bradshaw, "Information foraging as a foundation for code navigation (NIER Track)," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, Honolulu, HI, USA, May 2011, pp. 816–819.

[23] H. Qiao, Y. Li, T. Tang, and P. Wang, "Introducing memory and association mechanism into a biologically inspired visual model," *IEEE Trans. Cybern.*, vol. 44, no. 9, pp. 1485–1496, Sep. 2014.

[24] A. B. Özgüler and A. Yildiz, "Foraging swarms as Nash equilibria of dynamic games," *IEEE Trans. Cybern.*, vol. 44, no. 6, pp. 979–987, Jun. 2014.

[25] E. Nichols, L. McDaid, and N. H. Siddique, "Biologically inspired SNN for robot control," *IEEE Trans. Cybern.*, vol. 43, no. 1, pp. 115–128, Feb. 2013.

[26] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, MA, USA: Cambridge Univ. Press, 2008.

[27] E. H. Chi *et al.*, "The bloodhound project: Automating discovery of Web usability issues using the InfoScent simulator," in *Proc. Conf. Human Factors Comput. Syst. (CHI)*, Ft. Lauderdale, FL, USA, Apr. 2003, pp. 505–512.

[28] C. W. Bachman and R. G. Ross, "Toward a more complete reference model of computer-based information systems," *Comput. Stand.*, vol. 6, no. 5, pp. 331–343, Nov. 1982.

[29] X. Xu, Z. Huang, D. Graves, and W. Pedrycz, "A clustering-based graph Laplacian framework for value function approximation in reinforcement learning," *IEEE Trans. Cybern.*, vol. 44, no. 12, pp. 2613–2625, Dec. 2014.

[30] R. D. Baruah and P. P. Angelov, "DEC: Dynamically evolving clustering and its application to structure identification of evolving fuzzy models," *IEEE Trans. Cybern.*, vol. 44, no. 9, pp. 1619–1631, Sep. 2014.

[31] W. Gao, G. G. Yen, and S. Liu, "A cluster-based differential evolution with self-adaptive strategy for multimodal optimization," *IEEE Trans. Cybern.*, vol. 44, no. 8, pp. 1314–1327, Aug. 2014.

[32] C. M. Fernandes, A. M. Mora, J. J. Merelo, and A. C. Rosa, "KANTS: A stigmergic ant algorithm for cluster analysis and swarm art," *IEEE Trans. Cybern.*, vol. 44, no. 6, pp. 843–856, Jun. 2014.

[33] U. Halder, S. Das, and D. Maity, "A cluster-based differential evolution algorithm with external archive for optimization in dynamic environments," *IEEE Trans. Cybern.*, vol. 43, no. 3, pp. 881–897, Jun. 2013.

[34] H. Su *et al.*, "Decentralized adaptive pinning control for cluster synchronization of complex dynamical networks," *IEEE Trans. Cybern.*, vol. 43, no. 1, pp. 394–399, Feb. 2013.

[35] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.

[36] V. Tzerpos and R. C. Holt, "ADCD: An algorithm for comprehension-driven clustering," in *Proc. Working Conf. Reverse Eng. (WCRE)*, Brisbane, QLD, Australia, Nov. 2000, pp. 258–267.

[37] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Proc. Int. Conf. Program Compre. (ICPC)*, Kingston, ON, Canada, Jun. 2011, pp. 1–10.

[38] N. Niu and A. Mahmoud, "Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited," in *Proc. Int. Require. Eng. Conf. (RE)*, Chicago, IL, USA, Sep. 2012, pp. 81–90.

[39] N. Niu, L. D. Xu, and Z. Bi, "Enterprise information systems architecture—Analysis and evaluation," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2147–2154, Nov. 2013.

[40] W. Viriyasitavat, L. D. Xu, and W. Viriyasitavat, "Compliance checking for requirement-oriented service workflow interoperations," *IEEE Trans. Ind. Informat.*, vol. 10, no. 2, pp. 1469–1477, May 2014.

[41] N. Niu, L. D. Xu, J.-R. C. Cheng, and Z. Niu, "Analysis of architecturally significant requirements for enterprise systems," *IEEE Syst. J.*, vol. 8, no. 3, pp. 850–857, Sep. 2014.

[42] T. Bhowmik, N. Niu, A. Mahmoud, and J. Savolainen, "Automated support for combinational creativity in requirements engineering," in *Proc. Int. Require. Eng. Conf. (RE)*, Karlskrona, Sweden, Aug. 2014, pp. 243–252.

[43] J. R. Anderson, *The Adaptive Character of Thought*. Hillsdale, NJ, USA: Lawrence Erlbaum Assoc., 1990.

[44] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proc. Int. Workshop Program Compre. (IWPC)*, St. Louis, MO, USA, May 2005, pp. 259–268.

[45] A. E. Hassan and R. C. Holt, "Replaying development history to assess the effectiveness of change propagation tools," *Empir. Softw. Eng.*, vol. 11, no. 3, pp. 335–367, Sep. 2006.

[46] M. P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: An empirical study," *J. Softw. Maint. Evol. Res. Pract.*, vol. 22, no. 3, pp. 143–164, Apr. 2010.

[47] A. Mahmoud and N. Niu, "Evaluating software clustering algorithms in the context of program comprehension," in *Proc. Int. Conf. Program Compre. (ICPC)*, San Francisco, CA, USA, May 2013, pp. 162–171.

[48] R. K. Yin, *Case Study Research: Design and Methods*. Thousand Oaks, CA, USA: Sage, 2003.

[49] R. M. Sirkin, *Statistics for the Social Sciences*. Thousand Oaks, CA, USA: Sage, 2005.

[50] N. Niu and S. Easterbrook, "Extracting and modeling product line functional requirements," in *Proc. Int. Require. Eng. Conf. (RE)*, Barcelona, Spain, Sep. 2008, pp. 155–164.

[51] C. Xiao, "Using dynamic analysis to cluster large software systems," Master's thesis, Dept. Comput. Sci., York Univ., Toronto, ON, Canada, 2004.

[52] (Apr. 15, 2015). *Software Clustering Wiki*. [Online]. Available: http://wiki.cse.yorku.ca/project/cluster/start

[53] C. Duan and J. Cleland-Huang, "Clustering support for automated tracing," in *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*, Atlanta, GA, USA, Nov. 2007, pp. 244–253.

[54] Z. Wen and V. Tzerpos, "An optimal algorithm for MoJo distance," in *Proc. Int. Workshop Program Compre. (IWPC)*, Portland, OR, USA, May 2003, pp. 227–235.

[55] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 150–165, Feb. 2005.

[56] A. Mahmoud and N. Niu, "Source code indexing for automated tracing," in *Proc. Int. Workshop Trace. Emerg. Forms Softw. Eng. (TEFSE)*, Honolulu, HI, USA, May 2011, pp. 3–9.

[57] B. S. Mitchell and S. Mancoridis, “On the automatic modularization of software systems using the Bunch tool,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[58] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, “Opportunistic programming: Writing code to prototype, ideate, and discover,” *IEEE Softw.*, vol. 26, no. 5, pp. 18–24, Sep. 2009.

[59] A. T. T. Ying and M. P. Robillard, “The influence of the task on programmer behaviour,” in *Proc. Int. Conf. Program Compre. (ICPC)*, Kingston, ON, Canada, Jun. 2011, pp. 31–40.

[60] R. E. Brooks, “Towards a theory of the cognitive processes in computer programming,” *Int. J. Human Comput. Studies*, vol. 51, no. 2, pp. 197–211, Aug. 1999.

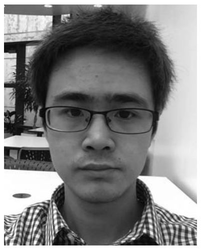
[61] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, “Cognitive design elements to support the construction of a mental model during software exploration,” *J. Syst. Softw.*, vol. 44, no. 3, pp. 171–185, Jan. 1999.

[62] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA, USA: MIT Press, 1981.

Nan Niu (M’08–SM’13) received the B.Eng. degree from the Beijing Institute of Technology, Beijing, China, the M.Sc. degree from the University of Alberta, Edmonton, AB, Canada, and the Ph.D. degree from the University of Toronto, Toronto, ON, Canada, all in computer science.

He is currently an Assistant Professor with the Department of Electrical Engineering and Computing Systems, University of Cincinnati, Cincinnati, OH, USA. His current research interests include software requirements engineering, information seeking in software engineering, and human-centered computing.

Dr. Niu is a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) Award.



Xiaoyu Jin received the B.Eng. degree from the Beijing University of Posts and Telecommunications, Beijing, China. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering and Computing Systems, University of Cincinnati, Cincinnati, OH, USA. His current research interests include software discovery, information retrieval, and data mining.

Zhendong Niu received the Ph.D. degree in computer science from the Beijing Institute of Technology, Beijing, China, in 1995.

He was a Post-Doctoral Researcher with the University of Pittsburgh, Pittsburgh, PA, USA, from 1996 to 1998, a Research/Adjunct Faculty Member with Carnegie Mellon University, Pittsburgh, from 1999 to 2004, and a Joint Research Professor with Information School, University of Pittsburgh, in 2006. He is a Professor and the Deputy Dean with the School of Computer Science and Technology, Beijing Institute of Technology. His current research interests include informational retrieval, software architecture, digital libraries, and Web-based learning techniques.

Dr. Niu is a recipient of the IBM Faculty Innovation Award in 2005 and the New Century Excellent Talents in University of Ministry of Education of China in 2006.

Jing-Ru C. Cheng received the Ph.D. degree in computer science from Pennsylvania State University, State College, PA, USA, in 2002.

She has been a Computer Scientist with the U.S. Army Engineer Research and Development Center, Vicksburg, MS, USA, since 2002. Her current research interests include parallel algorithm development, software tool development for scientific computing, and multiscale multiphysics code development.

Ling Li received the master’s and doctorate degrees in production/operations and logistics from the Ohio State University, Columbus, OH, USA, in 1994 and 1996, respectively.

She is a Professor of Production/Operations with Old Dominion University, Norfolk, VA, USA.

Dr. Li has served as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and other IEEE journals. She is a fellow in Production and Inventory Management, Association for Operations Management, Chicago, IL, USA.

Mikhail Yu Kataev received the M.S. degree in radio-physics engineering from the Tomsk State University, Tomsk, Russia, the Ph.D. degree in specialty optics from the Institute of Atmospheric Optics SB RAS, Tomsk, and the Dr.Sci. degree in specialty mathematical modeling, numerical methods and complexes of programs from the Tomsk State University of Control Systems and Radioelectronics (TUSUR), Tomsk, in 1984, 1993, and 2002, respectively.

He is currently a Professor with the Department Control Systems, TUSUR, and has been the Director of the Education Science Center since 2004. Since 2009, he has been the Vice Chief of the Chair Automated Control Systems. He is a Corresponding Member of the Russian Education Academy, Siberian Brunch. His current research interests include software engineering, numerical algorithms, parallel programming in atmosphere optics, and business process. He has published over 180 scientific papers and books in the field of solving forward and inverse tasks, recognition, and ill-posed.