Clustering and Recommending Collections of Code Relevant to Tasks

Seonah Lee and Sungwon Kang
Department of Computer Science
KAIST
Daejeon, Republic of Korea
{saleese, sungwon.kang}@kaist.ac.kr

***Abstract*—When performing software evolution tasks, programmers spend a significant amount of time exploring the code base to find methods, fields or classes that are relevant to the task at hand. We propose a new clustering approach called *NavClus* to recommend collections of code relevant to tasks. By gradually aggregating navigation sequences from programmers' interaction history, NavClus clusters pieces of code that are contextually related. The resulting clusters become the basis for NavClus to recommend collections of code that are likely to be relevant to the programmer's given task. We compare NavClus and TeamTracks, the state of the art code recommender for sharing navigation data among programmers. The results show that NavClus recommends pieces of code relevant to tasks considerably better than TeamTracks.**

Program comprehension; code navigation; data clustering techniques; context aware recommendation systems

I. INTRODUCTION

In order to make required changes to software systems as they evolve, programmers spend a significant amount of time—often higher than 50%—understanding the existing code base [7]. Despite such significant efforts, they often reach an impasse and must ask other programmers to explain the code base. However, such disruptive questions take other programmers away from their own tasks. As programmers are added to a project, the related communication overhead increases dramatically, and thus decreases programmers' productivity [1]. During program comprehension activities, programmers mainly explore the code base to identify the pieces of code that are relevant to their tasks [8]. If programmers are provided with the pieces of code that are relevant to their tasks, they will be able to reduce the time they spend on their evolution tasks and will less frequently interrupt other programmers. The overall productivity of programmers can thereby be improved.

Previous research has suggested recommending pieces of code relevant to tasks, based upon the history of programmers' activities in evolving software systems. This research can be divided into two classes. The first determines associations between pieces of code. Gall and colleagues initially proposed that if two pieces of code were changed together in the past, those pieces are highly likely to be changed together in the future [4]. They called this relationship logical coupling. Utilizing this logical coupling, Zimmermann and colleagues proposed mining association rules for recommending pieces of code to be changed together [13]. Likewise, DeLine and colleagues proposed mining consecutive visits between two pieces of code in the programmers' interaction history and using this information to recommend the pieces of code to be visited subsequently [3]. These approaches [3][13] are limited to recommending co-visited or co-changed pieces of code; many of the recommendations turn out to be irrelevant or not highly relevant to the given task. Robillard and Dagenairs [11] applied a data clustering technique to software revision history. However, their clusters are focused on the common parts of change sets and are not easily related to specific tasks. The second counts the frequencies of pieces of code over a period of time. Kersten and Murphy proposed supporting a programmer to collect pieces of code relevant to a task by automatically calculating a degree-of-interest [6]. Once a programmer identifies the starting point of a task, a degree-of-interest model counts the number of programmer interactions with pieces of code during the task. When a programmer addresses a task ID, the information is used to reveal a collection of code relevant to the task. As this approach relies on a programmer's manual identification of a task, a collection of code could comprise the contexts of two or more tasks. To address this, Coman and Sillitti proposed an approach that automatically separates task sessions [2]. However, in order to reveal task relevant code elements, a programmer is still required to know a specific task ID. The prior knowledge about the task ID impedes other programmers from reusing collections of code relevant to tasks.

Based upon the previous research, this paper focuses on increasing the task relevancy of code recommendations while minimizing programmers' required prior knowledge and effort. In a situation where more than two programmers perform similar tasks, if subsequent programmers are automatically given collections of code relevant to tasks from previous programmers' interaction history, they will be able to identify pieces of code that are relevant to their tasks with less effort. To that end, this paper proposes *NavClus* a novel approach that automatically clusters pieces of code that are contextually related. NavClus segments the programmers' interaction history into small navigation sequences. By gradually aggregating navigation sequences, NavClus counts the frequencies of code elements and determines contextual

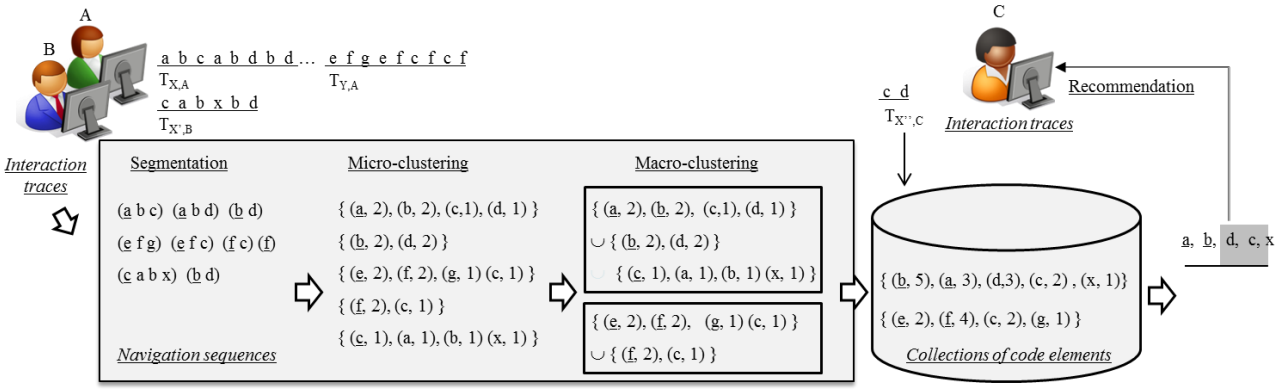

Figure 1. The NavClus' process of clustering and recommending collections of code

associations between code elements. As a result, NavClus produces collections of code that have been frequently navigated in similar contexts.

II. Task Relevance

To clarify what pieces of code should be clustered, we define the task relevance of a code element. A code element in this paper is a unit of arranging program statements, e.g. methods, fields or classes. A code element is relevant to an evolution task, if it is needed by a programmer who performs the evolution task. We classify the task relevance of a code element into four degrees: *A significantly relevant element [S]* is an element that a programmer needs to change in order to accomplish the task. *A highly relevant element [H]* is an element that a programmer needs to understand in order to know why and how to change a significantly relevant code element. *A fairly relevant element [M]* is an element that a programmer does not need to see but can be useful for understanding the context of a significantly or highly relevant code element. *A marginally relevant element [L]* is an element that a programmer does not need to see but is likely to read, because it exists in the same file that contains a significantly, highly or fairly relevant element. We propose two principles that help identify significantly or highly relevant code elements from the programmers' interaction history.

A. Relevance by Frequency

We define the frequency of a code element as the number of visits that programmers made to that code element. Researchers [3][6] have reported that the more frequently a code element is visited, the more importance it has. To confirm this, we counted the frequencies of code elements in the interaction histories of programmers, generated by an earlier study [12], and found that frequently visited code elements tend to be significantly or highly relevant to a certain task. Thus, we adopt *Principle 1: The code elements that programmers frequently visited are likely to be highly relevant to the tasks of the programmers.*

B. Relevance by Context

The context of a code element denotes a set of code elements that programmers navigate around that element. To identify the context of a code element, we use small sequences of code elements in which no identical element occurs twice. When a programmer returns to a code element in the programmer's navigation path, the previous sequence ends, and a new sequence begins at the code element. Through experiments on the interaction histories in [12] we found that within a navigation sequence the context of a code element also tends to be relevant to the same task to which the element is highly relevant. Thus, we adopt *Principle 2: If a code element of a navigation sequence is highly relevant to a task, it is likely that the other code elements in the same navigation sequence are relevant to the same task.*

III. NavClus

Based upon the two principles introduced in Section II, we propose *NavClus*, an approach to cluster collections of code elements that are contextually related. Fig. 1 presents an overview of NavClus with the scenario in which programmers A, B, and C perform similar tasks $T_{X,A}$, $T_{X',B}$, and $T_{X'',C}$. These code elements are represented with lower case letters in Fig. 1. NavClus clusters collections of code elements in the interaction histories where A and B performed $T_{X,A}$ and $T_{X',B}$, and recommends the code elements of $T_{X,A}$ and $T_{X',B}$ to C who is performing $T_{X'',C}$. The four steps of NavClus are presented in the following sub-sections.

A. Segmentation

This step segments a programmers' interaction history into small navigation sequences. A navigation sequence is a sequence of code elements in which an identical element does not occur twice. In the scenario, while performing $T_{X,A}$, programmer A navigated code elements a, b, c, a, b, d, b and d in that order. As programmer A visited a, b, c and returned to a, (a, b, c) becomes the first navigation sequence. Similarly, (a, b, d) and (b, d) are obtained as the second and third sequences, respectively. Therefore, the interaction history is segmented into three navigation sequences, (a, b, c), (a, b, d), and (b, d).

When a programmer's navigation path is represented as a graph, a code element that a programmer returns to is an intersection point of different navigation paths or a starting point of navigation loops. This point is an approximate center around which a programmer navigates, and it is likely to be a

code element that is frequently visited by the programmer. This point becomes the first element of a navigation sequence.

B. Micro-clustering

This step collects the navigation sequences that have the same first element. By Principle 1, the first element is likely to be highly relevant to a task. By Principle 2, the other elements of those sequences are highly likely to be relevant to the same task. Thus, this step produces a collection of code elements that are highly likely to be relevant to the same task. This collection represents the context of a high-frequency element. We call this a micro-cluster.

For example, the navigation sequences (a, b, c) and (a, b, d) are collected into a micro-cluster {(a, b, c), (a, b, d)} because they have the same first element. The navigation sequence (b, d) is not collected with them, because it has a different first element. In order to record the frequency of elements, the micro-cluster is transformed into another representation of a micro-cluster, which is a set of pairs of the form (element, frequency). Hence, in the above example, the micro-cluster {(a, b, c), (a, b, d)} is transformed into the micro-cluster {(a, 2), (b, 2), (c, 1), (d, 1)}.

C. Macro-clustering

This step groups similar micro-clusters generated from the previous step. By Principles 1 and 2, similar micro-clusters, where the same elements were frequently visited, are likely to be relevant to the same task. Thus, this step produces a collection of code elements that were frequently visited in similar contexts; we refer to this collection as a macro-cluster.

To measure the similarity between two micro-clusters, this step employs the cosine similarity, whose formula is A • B / ||A|| ||B||, where A and B are vectors. For example, two micro-clusters {(a, 2), (b, 2), (c, 1), (d, 1)} and {(b, 2), {d, 2}} can be expressed as vectors A = [2, 2, 1, 1] and B = [0, 2, 0, 2], and thus their cosine similarity becomes 0.67. As the value is close to 1, they are determined to be similar.

To group similar micro-clusters, this step ranks micro-clusters based on the frequency of their elements and compares lower ranked clusters to the top ranked cluster. For example, compared to {(a, 2), (b, 2), (c, 1), (d, 1)}, lower ranked clusters {(b, 2), {d, 2}} and {(c, 1), (a, 1), (b, 1), (x, 1)} yield cosine similarities 0.67 and 0.79. Then these micro-clusters are grouped into a macro-cluster {(b, 5), (a, 3), (d,3), (c, 2), (x, 1)}.

D. Recommendation

As a programmer interacts with the code base, NavClus retrieves a collection that contains the greatest number of elements that a programmer has recently visited. It then recommends code elements that have high frequencies in the collection. When programmer C navigates c and d, the collection {(b, 5), (a, 3), (d,3), (c, 2} is chosen, as it contains both c and d. NavClus recommends a and b, which have high frequencies in the collection. These are likely to be the relevant code elements that programmer C needs to navigate for the given task $T_{X'',C}$.

IV. Task Relevancy of NavClus' Recommendation

To evaluate NavClus, we measured the task relevancy of its recommendations, and compared the results to those of TeamTracks [3], which is the first approach developed for sharing navigation data among programmers and still the state of the art in recommending code elements to visit based on programmers' interaction history. For this evaluation, we used the interaction histories gathered in an earlier study [12], where twelve programmers performed four tasks. To cluster collections of code elements, we used the interaction histories of the first eight programmers. To simulate a programmer's interactions with the code base, we used the interaction histories of the last four programmers. The simulator made a recommendation based upon the 10 code elements that a programmer initially navigated to perform tasks.

To measure the task relevancy of NavClus' recommendations, we use Cumulated Gain (CG) [5]. CG is as commonly used to measure the effectiveness of information retrieval techniques. CG assumes that highly relevant items are more useful than marginally relevant items, which are more useful than irrelevant items. In CG, each item has a value that is predefined according to its degree of relevance, and CG cumulates the values of recommended items from the top to the n^{th} ranked one. If a system makes recommendations t times, the CG is averaged over the t recommendations. The formula for calculating the CG is: $CG = \sum_{j=1}^{t}\left(\sum_{i=1}^{n} x_{ij}\right) \Big/ t$

To calculate the CG of code recommendations, we first analyze the four tasks and identify the code elements relevant to each task. The elements relevant to Task 2-Arrows are presented in Table I. The code elements are identified with a three-character combination: the first character represents the relevance of the code elements, S, H, M or L. The second character represents tasks 1, 2, 3 or 4. The third is a unique character given to the task. We then assign the value of 2^d to those code elements according to its degree of task relevance.

- A significantly relevant element [S] ..... 2^3
- A highly relevant element [H] ..... 2^2
- A fairly relevant element [M] ..... 2^1
- A marginally relevant element [L] ..... 2^0

If code elements specified in Table I are recommended in the order of relevance from the most significantly relevant element first to the marginally relevant last <S2a, H2b, H2c,

TABLE I. Code elements relevant to task 2-Arrows that changes an arrow tip according to a menu

Relevance	Code Element	ID
Significantly relevant	PolyLineFigure.setAttribute()	S2a
Highly relevant	ChangeAttributeCommand.execute()	H2b
	ChangeAttributeCommand.ChangeAttribute-Command()	H2c
	DrawApplication.createArrowMenu()	H2d
	PolyLineFigure.ARROW_TIP_NONE (or BOTH)	H2e
	PolyLineFigure.ARROW_TIP_START (or END)	H2f
Marginally relevant	ArrowTip.ArrowTip()	M2g
	DrawApplication.createMenu()	M2h[1]
	DrawApplication.createAttributesMenu()	M2i[1]
	Figure.setAttribute()	M2j[1]

H2d, H2e, H2f, M2g, M2h, M2i, M2j, …>, this is regarded as an ideal recommendation. In this case, the CG becomes <8, 12, 16, 20, 24, 28, 30, 32, 34, 36 > as n increases from 1 to 10 (Ideal CG in Table II and Fig. 2).

From the interaction traces of the first eight programmers, NavClus generates seven collections of code elements. With the ten code elements that each of the last four programmers initially navigated to perform Task 2, NavClus retrieves the same collection that contains the code elements relevant to Task 2. As shown in Table II, NavClus recommends <H2b, H2d, S2a, H2c, L2s, M2o, M2k, M2m, L2x, L2y> in that order. Almost all recommended elements are relevant to Task 2. As a result, the obtained CG is <4, 8, 12, 20, 22, 26, 28, 30, 34, 35> (NavClus in Fig. 2).

As TeamTracks [3] has two modes of recommendation, we calculate two CGs. First, its view presents the code elements most frequently visited by programmers. For example, its view recommends S2a in the 2nd rank, H2b in the 6th and H2d in the 8th. While the recommendation contains significant and highly relevant code elements, it also contains many other task irrelevant code elements. (The task irrelevant code elements are indicated by a dash mark "-" in the table.) Based on the most frequently visited code elements, the CG for Task 2 is <0, 8, 8, 8, 8, 12, 16, 16, 16, 16>. Second, its editor recommends code elements consecutively visited from a particular code element. For example, its editor recommends <H2b, H2c, L2s>. This recommendation contains a small number of code elements which are not significant relevant code elements. For a fair comparison, we count the code elements most consecutively visited from 10 code elements that each of the last four programmers initially navigated; the CG is <1.3, 3.3, 5.5, 7, 7.5, 7.5, 7.8, 9.8, 9.8, 10.5>. The two CGs are shown in Fig. 2.

In Fig. 2, the CG of NavClus is higher than that of the TeamTracks' and, when the rank is 4 or greater, higher than twice that of the TeamTracks'. This indicates that NavClus recommends code elements that are highly or significantly relevant to Task 2-Arrows much better than TeamTracks. With regard to the other three tasks, NavClus also shows consistently higher CGs than TeamTracks.

TABLE II. RECOMMENDATION RESULTS FOR TASK2-ARROWS

Rank	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Ideal CG	S2a	H2b	H2c	H2d	H2e	H2f	M2g	M2h	M2i	M2j
NavClus	H2b	H2d	S2a	H2c	L2s	M2o	M2k	M2m	L2x	L2y
V.Team-Tracks	-	S2a	-	-	-	H2b	-	H2d	-	-
E.Team-Tracks	H2b	H2c	L2s							

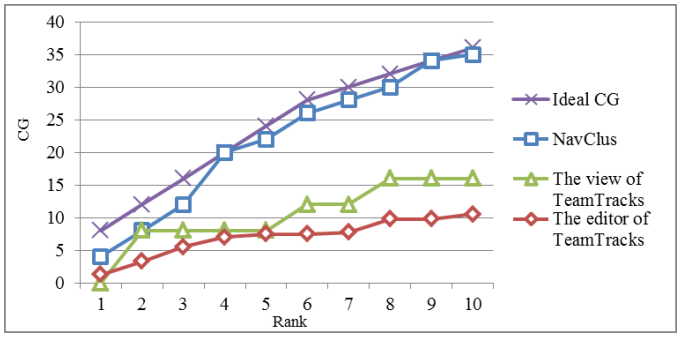

Figure 2. The recommendation results for Task 2, measured by CG.

V. CONCLUSION

We proposed NavClus, a novel approach that clusters contextually related code elements that are relevant to a programmer's given task. We also demonstrated that this approach recommends code elements relevant to tasks with high task relevancy by simulating it with the interaction traces gathered in a prior study [12]. The task relevancy of code recommendations by NavClus is consistently higher than that of TeamTracks'. We conducted a user study of this approach on the same interaction traces in another paper [9]. In the study, the participants evaluated that these recommendations are very helpful in finding relevant code elements. The preliminary evaluation with the real data obtained from the Eclipse Bugzilla system is reported in [10]. With the real data, it was demonstrated that NavClus makes useful code recommendations by calculating the similarity between clusters formed from programmers' past interaction history and a subsequent programmer's interactions with the code base.

We plan to extend this work by conducting more evaluations and investigating ways to manage clusters for more effective recommendations. Our ultimate goal is to automate transfer of code knowledge from programmer to programmer. To achieve this, we will research techniques for automatically organizing and presenting programmers' code knowledge.

REFERENCES

[1] F.P. Brooks JR, *The mythical man-month (anniversary ed.)*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[2] I. Coman and A. Sillitti, "Automated segmentation of development sessions into task-related subsections," *International Journal of Computers and Applications*, vol. 31, 2009, p. 159–166.

[3] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," *Proc. of VL/HCC*, 2005, pp. 241-248.

[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," *Proc. of ICSM*, 1998, pp. 190-198.

[5] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM TOIS*, 2002, pp. 422-446.

[6] M. Kersten and G.C. Murphy, "Using task context to improve programmer productivity," *Proc. of FSE*, 2006, pp. 1-11.

[7] A.J. Ko, H.H. Aung, and B.A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," *Proc. of ICSE*, 2005, pp. 126-135.

[8] T.D. LaToza and B.A. Myers, "Developers ask reachability questions," *Proc. of ICSE.*, 2010, pp. 185-194.

[9] S. Lee and S. Kang, "A study on guiding programmers' code navigation with a graphical code recommender," SpringerSCI, 2011, To appear.

[10] S. Lee and S. Kang, "Clustering and recommending collections of code relevant to tasks," CS-TR-2011-347, KAIST CS Department.

[11] M.P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, 2010, pp. 143-164.

[12] I. Safer and G.C. Murphy, "Comparing episodic and semantic interfaces for task boundary identification," *Proc. of CASCON*, 2007, pp. 229-243.

[13] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE TSE.*, 2005, pp. 429-445.