Clustering Navigation Sequences to Create Contexts for Guiding Code Navigation

Abstract

To guide programmer code navigation, previous approaches such as TeamTracks recommend pieces of code to visit by mining the associations between pieces of code in programmer interaction histories. However, these result in low recommendation accuracy. To create more accurate recommendations, we propose NavClus an approach that clusters navigation sequences from programmer interaction histories. NavClus automatically forms collections of code that are relevant to the tasks performed by programmers, and then retrieves the collections best matched to a programmer's current navigation path. This makes it possible to recommend the collections of code that are relevant to the programmer's given task. We compare NavClus' recommendation accuracy with TeamTracks' by simulating recommendations using 4,397 interaction histories. The comparative experiment shows that the recommendation accuracy of NavClus is twice as high as that of TeamTracks.

Highlights:

- To effectively guide code navigation, we propose a new approach that mines collections of code that are relevant to tasks.
- The approach clusters navigation sequences from programmers' interaction histories
- The approach uses the clusters to recommend the program elements that are relevant to a programmer's task at hand.
- NavClus' recommendation accuracy is compared with that of TeamTracks' using 4,397 interaction histories.
- The comparative experiment shows that the recommendation accuracy of NavClus is twice as high as that of TeamTracks.

Keywords: code navigation, programmer interaction histories, data clustering techniques, data stream mining, context aware code recommender.

1. Introduction

While making changes to a software system, programmers spend 35% of their time navigating the code base (Ko et al. 2006). Given a software maintenance task[1] such as fixing bugs or enhancing features, programmers navigate the code base to find relevant methods, fields or classes for accomplishing the task at hand. They then gather these pieces of code, and finally determine which pieces should be edited for the task (Ko et al. 2006; LaToza et al. 2010). Moreover, they perform multiple tasks concurrently, frequently switching between tasks unconsciously while often being interrupted by others (Mark et al. 2005), which makes the maintenance work much more challenging.

To guide programmers who are navigating the code base, programmer interaction histories—which include the records of programmers' edits and visits—can play an instrumental role, although previously non-visited navigation paths may have to be considered. Recommendations based on programmer interaction histories could help a programmer performs similar tasks that s/he or others did in the past. For example, when a programmer could not remember the details of the code base, s/he can be reminded of where s/he visited. These recommendations also

[1] When we mention "task" in this paper, it means "software maintenance task." The term is defined is in Section 2.1.

could transfer one programmer's contextual knowledge to another programmer's. For example, when performing tasks similar to what others have done, the programmer will work efficiently by knowing what collections of code were navigated in the previous tasks.

Thus researchers have mined programmer interaction histories. For example, TeamTracks (DeLine et al. 2005) mines the consecutive visits between methods (functions) and recommends subsequent methods to visit, as a programmer navigates the code base. Likewise, by mining interaction histories, NavTracks (Singer et al. 2005) recommends subsequent source files to visit and Switch! (Sahm et al. 2010) recommends other artifacts to visit next. However, various user studies (DeLine et al. 2005; Singer et al. 2005; Robbes et al. 2010; Lee et al. 2011) show that these approaches yield low recommendation accuracy and thus do not effectively guide programmers' code navigation. For example, when a programmer performed an evolution task, TeamTracks recommended only three methods, none of which turned out to be needed for edit in order to accomplish the task (Lee et al. 2011). The main reason for the inaccuracy of the recommendations of these approaches is that they mine the associations between two pieces of code in a short distance.

To create accurate recommendations, we propose mining collections of code relevant to tasks in programmer interaction histories. However, mining programmer interaction history poses the following challenges::

•Programmer interaction history is a continuous data stream that is augmented as time passes. A data-mining technique that scans the interaction history several times will be impractical (Han et al. 2000; Masud et al. 2010).

•Programmer interaction history contains unseen spatial data. Because it is uncertain which source locations are close to each other, a data-mining technique that calculates a numerical distance between locations will be difficult to apply.

•Programmer interaction history shows no clear task boundaries. Programmers frequently switch their tasks unconsciously and a program element could be related to more than one task (or feature). A data-mining technique should not use boundary information but should allow the overlapping of collections of code to be mined.

To overcome these challenges, we propose a new approach *NavClus*. NavClus segments programmer interaction histories into navigation sequences and then clusters them. Our insight is that a navigation sequence contains the program elements that are contextually related to each other. By collecting similar navigation sequences, NavClus recovers programmers' past navigation contexts, which are the collections of code that are relevant to tasks performed in the past. NavClus then retrieves a past navigation context that best matches a programmer's current navigation path and recommends a collections of code to the programmer.

To evaluate NavClus, we simulate the code recommendations using 4,397 interaction traces obtained from the Eclipse Bugzilla system[2]. We then compare the recommendation accuracy of NavClus with that of TeamTracks, a state-of-the-art code recommender for sharing navigation data among programmers (DeLine et al. 2005). Our evaluation demonstrates that NavClus yields recommendation accuracy that is twofold higher than that of TeamTracks.

We make the following contributions. First, we propose a clustering technique that automatically forms past programmers' navigation contexts by using both the associations and frequencies of program elements in programmer interaction histories. Second, we propose an approach called NavClus that recommends a navigation context that contains the greatest number of program elements currently being navigated by a programmer. Third,

[2] https://bugs.eclipse.org/bugs/

based on the approach, we implemented the NavClus tool, which yields much higher recommendation accuracy than the state-of-the-art approach TeamTracks (DeLine et al. 2005).

This paper is organized as follows. Section 2 discusses our idea of clustering navigation sequences to create navigation contexts. Section 3 explains the NavClus algorithm, which automatically clusters navigation sequences and recommends collections of code relevant to the task at hand. Section 4 describes our experiment design, and Section 5 reports the evaluation results. Section 6 discusses related work, and Section 7 concludes the paper.

2. Clustering Navigation Sequences

In this section, we first define the term "navigation context" (Section 2.1). We then present two principles that help identify these navigation contexts within programmer interaction histories (Section 2.2). We finally explain how navigation contexts are formed through the clustering of navigation sequences (Section 2.3).

2.1. Navigation Context

A software maintenance task is the identifiable and essential unit of work that changes a software system, after the system has been delivered. A software maintenance tasks can be the work of fixing bugs or enhancing features. While performing software maintenance tasks, programmers first understand the context of a program element, determine which program elements to next examine, identify the program elements relevant to their tasks, and finally edit program elements necessary for accomplishing their tasks (Ko et al. 2006). To define the collections of program elements that programmers need to visit during their code navigation activities for their tasks, we introduce the following notion:

Definition (Navigation Context): *Navigation context* is the information — a graph of program elements and the relationships between them — that a programmer needs to explore, gather and understand during a software maintenance task[3].

To guide a programmer's code navigation effectively, a recommendation system needs to provide a navigation context on the fly as a programmer navigates the code base. To do this, the recommendation system needs to create these navigation contexts in advance. Our conjecture is that the recommendation system can use the collections of program elements that programmers navigated in the past to automatically create navigation contexts.

2.2. Principles for Creating Navigation Contexts

To automatically create navigation contexts from programmer interaction histories, we propose two principles to identify collections of program elements relevant to tasks. A program element is relevant to a task, if it is needed by a programmer who performs the task. We classify the task relevance of a program element into four degrees: A significantly relevant element is an element that a programmer needs to change in order to accomplish the task. A highly relevant element is an element that a programmer needs to understand in order to know why and how to change a significantly relevant program element. A fairly relevant element 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 program element. A marginally relevant element is an element that a programmer does not need to see but is likely to read, because it

[3] This definition is derived from the concept of 'Task Context', defined by Kersten and Murphy as "the information — a graph of elements and relationships of software artifacts — that a programmer needs to know to complete the task" (Kersten et al. 2006).

exists in the same file that contains a significantly, highly or fairly relevant element. Our mining goal is to mine the collections of program elements that are significantly or highly relevant to tasks.

1) Relevance by Frequency

We define the frequency of a program element as the number of visits that programmers make to that program element. Researchers (DeLine et al. 2005; Kersten and Murphy 2006) have reported that the more frequently a program element is visited during a task, the greater its importance is in completing that task. To confirm this, we mined the interaction histories of programmers generated by an earlier study (Safer et al. 2007) and calculated the frequencies of the program elements. We found that frequently visited program elements tend to be significantly or highly relevant to a certain task. Thus, we adopt Principle 1:

Principle 1 (Relevance by Frequency): The program elements that programmers frequently visited are likely to be significantly or highly relevant to the tasks of the programmers.

2) Relevance by Association

We view the successive visits that programmers make from one program element to another as an association between them. Researchers (DeLine et al. 2005; Singer et al. 2005) have suggested that the more frequently two program elements are successively visited, the more contextually associated they are. To identify the association among program elements, we utilize the navigation sequence defined as follows.

Definition (Navigation Sequence): A *navigation sequence* is a sequence of program elements successively visited by a programmer; a navigation sequence ends where a programmer returns to a program element that s/he has already visited and a new sequence begins at the repeated program element.

Through experiments on the interaction histories in (Safer et al. 2007) we found that if an element of a navigation sequence is relevant to a task then the other program elements in it also tends to be relevant to the same task. Thus, we adopt Principle 2:

Principle 2 (Relevance by Association): If a program element of a navigation sequence is relevant to a task, it is likely that the other program elements in it are relevant to the same task.

2.3. Clustering Navigation Sequences to Create Navigation Contexts

To recommend collections of program elements that are relevant to a programmer's task at hand, we intend to create navigation contexts. To achieve this, we need to integrate the collections of code relevant to similar tasks and to separate the collections of code relevant to dissimilar tasks. To meet this need, we propose clustering navigation sequences. To explain our idea, we first define the following key terms.

Definition (Similar Tasks): Two tasks are similar if programmers modify the same feature[4] by using the same software artifacts in the same development environment.

Definition (Similar Navigation Sequences): Two navigation sequences are similar if they contain more of the same program elements than different program elements.

[4] Features are the characteristics of software systems visible to users.

To determine if two navigation sequences are similar, we can use a similarity metric. For example, if we use the cosine similarity metric, Va • Vb / ||Va|| ||Vb|| where Va and Vb are vectors (Han et al. 2000), navigation sequences (a, b, c) and (a, b, d) are converted to two vectors (1, 1, 1, 0) and (1, 1, 0, 1) and then their cosine similarity is calculated as 2/3 = 0.66. If their cosine similarity is greater than a certain threshold (for example, a threshold of 0.5), the navigation sequences (a, b, c) and (a, b, d) can be determined to be similar.

Our insight is that clustering similar sequences can form collections of code relevant to similar tasks. According to principle 2, program elements in a navigation sequence are likely to be relevant to a task. Let α and β be navigation sequences created from the program elements programmers navigated while performing certain tasks, say Ta and Tb. If α and β share elements, the elements in α are likely to be relevant to Tb (and likewise elements in β are likely to be relevant to Ta). The more elements two navigation sequences α and β share, the more likely the elements in α are relevant to task Tb and vice versa.

Suppose that a programmer performs two dissimilar tasks, Tx and Ty, and performs the task Tx' that is similar to task Tx. While performing task Tx, a programmer sequentially navigates the program elements that are represented as small letters: (a, b, c, a, b, d, b, d). Later, the programmer then performs task Ty and navigates program elements (e, f, g, e, f, c, f, c), and then performs task Tx' and navigates program elements (c, a, b, x, b, d).

In this situation, the system just observes a sequence of events. The system thus observes a sequence of events that represent programmers' visits to the program elements as follows:

a, b, c, a, b, d, b, d, ∨, e, f, g, e, f, c, f, c, c, a, b, x, b, d

The symbol ∨ in the above sequence represents a time point when a programmer stopped for a while. Using the definition from Section 2.2 our approach segments this sequence of events and creates navigation sequences as follows:

(a, b, c), (a, b, d), (b, d), (e, f, g), (e, f, c), (f, c), (c, a, b, x), (b, d)

We then cluster similar navigation sequences by putting a navigation sequence into a group where one of the previous navigation sequences is most similar to sequence α such that the similarity is over a threshold (for example, 0.5). The first two navigation sequences (a, b, c) and (a, b, d) are grouped into group Ga because their similarity value is 0.66. The navigation sequence (b, d) is added into the same group Ga because the similarity value of (b, d) and (a, b, d) is 0.82. In this way, we can cluster the above navigation sequences into two groups:

{(a, b, c), (a, b, d), (b, d), (c, a, b, x), (b, d)}, … {(e, f, g), (e, f, c), (f, c)}
Ga Gb

We now introduce a term that is central to our approach.

Definition (Contextually Related): Program elements are contextually related if they are in two similar navigation sequences.

For example, the program elements a, b, c and d in the two similar navigation sequences (a, b, c) and (a, b, d) are contextually related. Once we cluster sequences into two groups of program elements that are contextually related, we then identify the program elements that have significance by counting their frequencies based on principle 1.

{(a, 3), (b, 5), (c, 2), (d, 3), (x, 1)}, … {(e, 2), (f, 3), (g, 1), (c, 2)}

Ga Gb

The group Ga consisting of a, b, c, d, and x represents the navigation context of Tx and Tx'. The group Gb consisting of c, e, f and g represents the navigation context of Ty. We call these groups *navigation contexts*.

These navigation contexts will be used to guide programmers' code navigation by recommending more program elements that are significant to the current task of a programmer. For example, if programmer C performs Tx' and navigates c and d, the programmer will receive a recommendation of a and b which were frequently visited through tasks Tx and Tx'. The program element x will be excluded from the recommendation because it has a low visit frequency.

3. The NavClus Algorithm

Mining collections of code relevant to tasks in programmer interaction histories is challenging, because of the characteristics of programmer interaction histories, described in Section 1. To overcome this challenge, we surveyed data clustering techniques and decided to adopt the following two strategies:

Strategy 1 (Partitioning & Grouping): Segment an interaction trace into smaller navigation sequences and collect similar navigation sequences into groups.

The objective of Strategy 1 is to mine fine-grained navigation contexts. The strategy is similar to the partition and group framework proposed by Lee et al. for mining fine-grained trajectory paths (Lee et al. 2007). Partitioning a trajectory path into small lines and grouping small lines into a standardized line reveals a more accurate trajectory path than existing trajectory clustering approaches which simply standardize the entire trajectory paths. The adoption of this strategy helps to mine the collections of code that are relevant to tasks more accurately than previous approaches (DeLine et al. 2005; Robillard et al. 2010).

Strategy 2 (Micro-clustering & Macro-clustering): As a data stream of programmer interactions comes in, create navigation sequences and group these navigation sequences into small sets of navigation sequences. Group these sets of navigation sequences into navigation contexts.

The strategy of micro-clustering and macro-clustering was first proposed by Aggarwal et al. with the objective of mining the data stream efficiently (Aggarwal et al. 2003). Micro-clusters are the summaries that are instantly formed when a data stream comes into a system. Macro-clusters are the summaries that are formed from micro-clusters. Maintaining micro-clusters and macro-clusters separately allows the system to scan the entire data stream once while continually updating the final summaries.

With these two strategies, we propose our approach, NavClus. It has two phases: (1) the mining phase, followed by (2) the recommendation phase. The mining phase is divided into the segmentation step and the clustering step by using Strategy 1. The clustering step is divided into the micro-clustering step and the macro-clustering step by using Strategy 2. NavClus therefore consists of four steps: Segmentation, Micro-clustering, Macro-clustering and Recommendation.

We present an outline of the NavClus algorithm in Fig. 1. The three steps of the mining phase sequentially process a sequence of interaction events to update the navigation contexts in a database. The recommendation phase retrieves the navigation contexts from the DB using the current programmer's interaction trace and recommends program elements relevant to the task at hand. To trigger a recommendation, it uses k number of program elements that a

programmer is currently navigating. If it finds the navigation contexts that include these program elements, it recommends the navigation contexts.

Algorithm NavClus

INPUT: sequence of interaction events $S = (ie_1, ie_2, \cdots, ie_{ns})$
OUTPUT: list of recommended elements $R = [e_1, e_2, \cdots, e_{nr}]$
VARIABLE: (1) set of navigation contexts $F = \{f_1, f_2, \cdots, f_{nf}\}$
(2) k-length queue of navigated elements $P = [[e_1, e_2, \cdots, e_{np}]]$

```
ALGORITHM:
for each (ie ∈ S) {

  /* MINING PHASE */
  F.update(macro-cluster(micro-cluster(segment(ie))));

  /* RECOMMENDATION PHASE */
  P.update(ie);
  if (P.isfull() ){
    R = F.retrieve(P);
  end if
}
```

Fig. 1. The main algorithm

3.1. Segmentation

This segmentation step aims to partition a programmer's interaction history into small navigation sequences in order to maintain both visit frequencies of program elements and visit associations between them. This is a pre-processing step to identify program elements that have significance in a context by Principle 1 as well as to identify a set of program elements that belongs to the same context by Principle 2.

To segment a programmer interaction history into navigation sequences, we use the definition of a navigation sequence in Section 2.2. When a programmer returns to a program element that s/he has already visited, one navigation sequence ends and a new navigation sequence begins (Singer et al. 2005). Every program element of a navigation sequence thus has a visit frequency of 1, and the first element of the navigation sequence becomes a return point. When a programmer's navigation path is represented as a graph, a program element – to which a programmer returns – becomes an intersection point of different navigation paths or a starting point of a navigation loop. This point is an approximate center around which a programmer navigates, and it is likely to be a program element that is frequently visited by the programmer (Principle 1).

The input for this step is the sequence of interaction events (S) being recorded while programmers interacted with the code base. The output of the step is a set of navigation sequences. Each navigation sequence (NS) is a sequence of program elements ($e_1, e_2, \cdots, e_{nn}$) in which no identical element occurs twice. When receiving an interaction event (ie) from S, this step adds the element e of ie to the navigation sequence (NS). If NS already contains e, this step segments the interaction history: if ($e \in$ NS) then segment NS.

Example (Segmentation): A programmer interaction history records interactions with program elements a, b, c, a, b, and d in time order. Because the program element "a" at the first position reappears at the fourth position, the first three elements a, b and c, become the first navigation sequence (a, b, c).

3.2. Micro-clustering

This micro-clustering step aims to form the context of a frequently visited element. Here, the context of a program element is a collection of other elements navigated from that program element. The frequently visited element has significance in a context (Principle 1), and its context can be used to identify the navigation contexts to which the program element may belong (Principle 2).

The micro-clustering step uses the segmented navigation sequences to identify the contexts of those program elements. To identify the context of a particular program element, this step collects navigation sequences that have the same first element. The first program element is likely to be the program element that was more frequently visited by programmers. The other elements in the navigation sequences become the context of the frequently visited program element. To store this type of collection, we define a micro-cluster as follows:

Definition (Micro-cluster): A *micro-cluster* is a collection of program elements that were navigated around a particular program element. A micro-cluster is represented as a set of pairs of the form (e, f), where e is a program element and f is its visit frequency, as accumulated within the cluster.

The inputs of the step are the segmented navigation sequences generated by the previous step. The first program element in each sequence e_1 is defined as the head element of each navigation sequence, NS.HEAD. The output of the step is micro-clusters where a micro-cluster MS is a bag of program elements $\{(e_1, n_1), (e_2,n_2)\cdots, (e_{n\mu}, n_{n\mu})\}$. Each micro-cluster MS also has its own head element MS.HEAD. When receiving navigation sequences, this step finds a micro-cluster that has the same head element as the navigation sequence. If it finds such a micro-cluster, it records the navigation sequence in association with that micro-cluster: if (MS.*head* == *NS.head*) and then adds NS to MS. If not, it creates a new micro-cluster with the new head element to store the navigation sequence.

Example (Micro-clustering): 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 elements. The micro-cluster {(a, b, c), (a, b, d)} is transformed into the bag structure {(a, 2), (b, 2), (c, 1), (d, 1)}.

3.3. Macro-clustering

This macro-clustering step aims to collect the program elements that have been frequently visited in similar contexts. Similar contexts are likely to be highly relevant to similar tasks. Thus, collecting similar contexts (micro-clusters) are expected to produce collections of program elements that are relevant to similar tasks. To store these types of collections, we define a macro-cluster as follows:

Definition (Macro-cluster): A *macro-cluster* is a collection of micro-clusters that are similar to each other. This collection can identify a set of program elements that has high frequencies in similar contexts. A macro-cluster is represented as a set of pairs of the form (e, f), where e is a program element and f is its visit frequency as accumulated within the cluster.

To measure the similarity between two micro-clusters, this step employs the cosine similarity metric, which is calculated as Va • Vb / ||Va|| ||Vb||, where Va and Vb 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]; thus, their cosine similarity becomes 0.67. If this value is larger than a similarity threshold θ_1, they are determined to be similar.

The inputs of the step are the micro-clusters. To collect similar micro-clusters, this step uses a clustering algorithm which we developed[5]. The algorithm is presented in Fig. 2. Step 1 of the algorithm first finds pairs of micro-clusters that are most similar to each other by calculating the cosine similarity. The similar pairs of clusters are the only necessary pieces of information to group similar micro-clusters. Step 2 of the algorithm sorts the pairs from those with the highest similarity value to those with the lowest similarity value. This is done to prepare seeds for the next clustering step. Step 3 of the algorithm next groups similar micro-clusters based upon the list of pairs. Step 4 of the algorithm finally creates macro-clusters C that represent programmers' past navigation contexts.

Algorithm Macro-clustering

INPUT: A set of micro-clusters where each micro-cluster MS is a bag of program elements {(e1, n1), (e2,n2), ... , ($e_{n\mu}$, $n_{n\mu}$)}

OUTPUT: A set of macro-clusters where each macro-cluster C is a bag of program elements {(e1, n1), (e2,n2), ... , (e_{nc}, n_{nc})}

PARAMETER: A similarity threshold where $0 \leq \theta_1 \leq 1$

ALGORITHM:

1. Find the most similar micro-cluster MS_j for each micro-cluster MS_i, and pair them (MS_i, MS_j) with the similarity value $\theta_{i,j}$ between them

2. Sort the pairs from the most similar pair to the least similar pair

3. Group similar micro-clusters based upon the sorted list of pairs, if the similarity is larger than θ_1.

```
for (each pair from the top rank in the sorted list of pairs) {
  for (each group) {
    if (a group contains MSj of the pair)
      pair.similarity = the similarity of the pair
      pair.group = this group
    }
  }
  if (pair.similarity ≥ θ1 ) {
    merge the pair to pair.group
  else {
    create a new group and merge the pair to the new group
  }
}
```

4. Create each macro-cluster for each group

Fig. 2. The macro-clustering algorithm

Example (Macro-clustering): Given five micro-clusters A:{(a, 2), (b, 2), (c, 1), (d, 1)}, B:{(b, 2), (d, 2)}, E:{(e, 2), (f, 2), (g, 1) (c, 1)}, F:{(f, 2), (c, 1)} and C: {(c, 1), (a, 1), (b, 1) (x, 1)}, each micro-cluster is paired with the most similar micro-cluster, and the pairs are sorted from highest similarity value to lowest value: (A, C), (C, A) (E, F), (F, E) and (B, A) in Step 1 and 2 of algorithm above. Similar micro-clusters based upon the sorted list of pairs are then grouped, only if the pairs have higher similarity than 0.5: {A, C, B} and {E, F} in Step 3 and 4. These two groups are converted to {(b, 5), (a, 3), (d, 3), (c, 2), (x, 1)} and {(e, 2), (f, 4), (c, 2), (g, 1)}.

3.4. Recommendation

[5] To group similar micro-clusters, we can alternatively employ the k-nearest neighbor algorithm [Robillard 2010] with a similarity threshold.

This recommendation step aims to recommend collections of program elements that are likely to be relevant to a task that a programmer is performing. To recommend these collections, it retrieves the macro-clusters that contain the greatest number of elements recently visited by the programmer.

To retrieve macro-clusters that contain the greatest number of elements that a programmer has recently visited, this step calculates the similarity between macro-clusters and the set of program elements that the programmer is navigating. To measure the similarity, the step employs the TF-IDF similarity metric (Han et al. 2000). In our application of TF-IDF, a document is a macro-cluster and a term is a program element of a macro-cluster. TF (Term Frequency) represents the number of occurrences of a program element in a macro-cluster, IDF (Inverse Document Frequency) is defined as log (N+n)/n, where N is the number of total macro-clusters and n is the number of macro-clusters containing a program element. By multiplying TF by IDF, the step can calculate the TF-IDF value of each program element in macro-clusters. It then measure the cosine similarity between macro-clusters and the set of program elements that the programmer is navigating by utilizing these TF-IDF values.

The inputs of this step are macro-clusters C and the current navigation path P. The data structure for P is a queue which holds k program elements. The queue maintains the program elements recently visited by a programmer, adding a new program element visited by the programmer and removing the oldest program element. Whenever P obtains a new program element, this step tries to create a recommendation. The step retrieves macro-clusters which are similar to the path P, by computing the TF-IDF values between each macro-cluster c and P. If it finds the macro-cluster whose similarity values are greater than a certain similarity threshold θ_2, it adds the macro-clusters to the recommendation list R. The step ranks program elements in the recommendation list according to their frequency, and finally recommends ten program elements that have high frequencies as a default number of elements.

Example (Recommendation): Given two macro-clusters {(b, 5), (a, 3), (d, 3), (c, 2), (x, 1)} and {(e, 2), (f, 4), (c, 2), (g, 1)} and the current navigation path (c, d), the macro-cluster {(b, 5), (a, 3), (d, 3), (c, 2), (x, 1)} is selected, because it contains both c and d, thus having higher TF/IDF value than the other cluster. Program elements a and b are then recommended, because they have high frequencies in the recommendation list.

4. Experimental Design

To evaluate NavClus, we chose a simulation methodology, which enabled us to compare two different recommendation methods under the same conditions. We simulated code recommendations of two different approaches: TeamTracks and NavClus. TeamTracks is a state-of-the-art approach that recommends subsequent program elements to visit for sharing navigation data among programmers.

4.1. Research Questions

The evaluation was performed to answer the following two questions:

Q1. Does the proposed approach, NavClus, recommend collections of code that are relevant to a task at hand more accurately than the state-of-the-art TeamTracks?

Q2. Do the NavClus recommendations include the program elements to be edited more frequently than the TeamTracks recommendations?

The first question (Q1) evaluates if the collections of code automatically created by NavClus help recommend the program elements that are relevant to a task at hand. Our hypothesis is that the proposed approach NavClus will form the collections of code that represent programmers' past navigation contexts and will thus recommend more program elements relevant to a task than the previous approach TeamTracks..

The second question (Q2) evaluates if the recommendations made by NavClus are more significant compared to those by TeamTracks. Not all of the program elements navigated by a programmer are highly relevant to a given task. The degree of task relevance of a program element varies (Lee et al. 2011). A significantly relevant element is a program element that a programmer needs to edit in order to make some changes. Thus, we count how many recommendations include program elements to edit in a given task. We check if NavClus makes such recommendations more frequently than TeamTracks.

4.2. Subjects

In this experiment, we used Mylyn data[6] extracted from the Eclipse Bugzilla system. Mylyn is integrated with Eclipse IDE, and it helps programmers to store their interaction traces into the Eclipse Bugzilla system (Kersten et al. 2006). However, Mylyn was only recently integrated with the Eclipse Bugzilla system; thus, the five projects in Table 1 have the highest number of interaction traces stored in the Eclipse Bugzilla system. We used the five projects in Table 1 as subjects.

Table 1. The interaction traces of five software projects

Projects	Description	Period	#Interaction traces	#views/ trace	#edits/ trace	View-edit-ratio
Mylyn	A task management tool for programmers	2006-05-18 ~2010-08-31	2,726	35.9	6.7	5.4
Eclipse Platform	Eclipse core frameworks	2008-02-18 ~2010-08-25	582	62.7	7.3	8.6
Eclipse PDE	For building Eclipse plug-ins	2007-11-11 ~2010-06-16	536	15.0	2.2	6.8
ECF	For building distributed servers & applications	2007-04-06 ~2010-08-08	308	15.6	1.4	11.1
MDT	For UML modeling	2010-01-26 ~2010-08-31	245	58.8	1.5	39.2

An interaction trace consists of records that contain details of programmer interaction events, including the starting date, the ending date, the type of event, and the program element. According to the type of event, the program elements can be classified into viewed elements and edited elements. If an interaction event has a “selection” event type, the event is classified as a view event. The program element in the interaction event is a viewed element. If an interaction event has the “edit” event type, the event is classified as an edit event. The program element in the interaction event is an edited element. There is an exception: if an interaction event has the same starting and ending date, the event is not classified as an edit even if it has an “edit” event type, as the event represents the exceptional case in which a programmer double-clicks on a program element and Mylyn subsequently records the “edit” typed event (Lee et al. 2011).

Table 1 also presents each project’s average number of viewed elements and edited elements, and the ratio of edited elements to viewed elements per interaction trace. The characteristics of interaction traces differ for each project. First, the numbers of interaction traces differ. For instance, the Mylyn project has 2,726 interaction traces because the Mylyn data has been accumulating since 2006. The MDT project has 245 interaction traces because the MDT data has been accumulating since 2010. Second, the ratios of edited elements to viewed elements (the view-edit-ratio) differ. For example, the Eclipse PDE project has a 6.8 view-edit ratio, which means that programmers edited 1 program element while visiting 7 program elements on average. The MDT project has a 39.2 view-edit ratio, which means that programmers edited 1 program element while visiting 40 program elements on average.

We sorted the records of the interaction traces in time order before the simulation, as the records of Mylyn interaction traces occasionally do not appear in chronological order (Ying et al. 2011). The monitoring facility of Mylyn (Kersten et al. 2006) aggregates the records of an interaction trace in order to compress its volume. We sorted the records according to the starting date of each interaction record and then stored the rearranged records into new data files. In this simulation, we used these rearranged records.

[6] To our best knowledge, Mylyn data is the only source of programmer interaction histories that have been accumulated over several years.

4.3. Methodology

We conducted a controlled comparative experiment and simulated code recommendations with the Mylyn data, as explained in Section 4.2.

In order to simulate code recommendations, we took an on-line learning approach that mines all of the past histories to make a recommendation at the current time point. For example, if we have N number of interaction traces that belong to a project, a simulator uses all of the interaction traces except the initial one in order to simulate programmer interactions. When the simulator makes recommendations from interaction trace T_i, the simulator mines the interaction traces that occurred prior to T_i (from T_1 to T_{i-1}).

Under the above condition, we reimplemented TeamTracks and used the recommendation results of TeamTracks as a baseline for our comparison. TeamTracks counts consecutive visits between two program elements. When a programmer visits an element, TeamTracks uses this information to recommend elements to visit next.

We also implemented NavClus and used the recommendation results of NavClus as a control group to show better results than TeamTracks. As NavClus holds few parameters, we set the parameters. During the macro-clustering step, we set the similarity threshold θ_1 to 0.5. In the recommendation step, we set the size of a query to 3, where the query consists of three program elements. We also set the similarity threshold θ_2 to 0.7.

4.4. Measurement

To evaluate our approach in terms of the two questions Q1 and Q2, we use four measures: Precision, Recall, F-measure, and the Edit-Hit ratio.

1) Precision, Recall, and F-measure

Precision and recall are common measures for evaluating the effectiveness of an information retrieval routine. To measure precision and recall, two sets of program elements are used: a set of recommended elements (A) and a set of expected elements (E). Set A includes the program elements recommended at each recommendation time. Set E includes the program elements that are viewed or edited elements in T_i. Both Sets A and E exclude the program elements used to make recommendations. F-measure is the harmonic mean of the precision and recall. Precision (P), recall (R) and F-measure (F) are computed as shown below.

$$P = |A \cap E| / |A| \qquad R = |A \cap E| / |E| \qquad F = (2*P*R) / (P+R)$$

It should be noted that the F-measure does not have a high value because of the recall. The cardinality of A, denoted as |A|, is always less than or equal to 10, because we only count the program elements that are ranked in the top 10. |E| is occasionally more than 100, because T_i occasionally includes more than a hundred elements. For instance, if |A| is 10, |E| is 100 and $|A \cap E|$ is 10, the precision P is 1.0 and the recall R is 0.1. In this case, F is 0.18. If we have many recommendations, we first obtain the average precision and the average recall. We then calculate the average F-measure.

2) Edit-Hit Ratio

We define the Edit-Hit Ratio as the ratio of recommendations that include the program elements to edit to the total recommendations. This is as follows:

EH = (The number of recommendations that include elements to edit) / (Total number of recommendations)

Here, the program elements to edit are the elements that are edited in T_i and the recommendations that include the program elements to edit are the recommendations that occurred in T_i.

5. Results and Analysis

We simulated the recommendations of NavClus and TeamTracks using the interaction traces of 5 projects. Based on the simulated results, we present the number of recommendations, and then evaluate the effectiveness of NavClus by comparing its recommendation results with those of TeamTracks. To evaluate the first question (Q1), we review the precision, recall, and F-measure values. To evaluate the second question (Q2), we review the edit-hit-ratio values.

We first present the number of recommendations that each approach produced, because the number of recommendations could also be related to the effectiveness of recommendation approaches. Even if an approach produces highly accurate recommendation results, if the approach rarely makes recommendations, users will not able to utilize the recommendations. We thus discuss if each approach produced enough recommendations.

Table 2 presents the number of recommendations that our simulation produced per subject. For instance, our simulation of TeamTracks generated 241,084 recommendations for the Mylyn project. Because the Mylyn project has the 2,726 interaction traces, the average number of recommendations per trace is 88. Our simulation of NavClus generated 137,979 recommendations for the Mylyn project. The average number of recommendations per trace is 51. NavClus made 43% less recommendations than TeamTracks. However, this difference is negligible. Because both approaches try to make a recommendation per visit, programmers will see their recommendations within a few visits.

Table 2. The number of code recommendations

Project	**TeamTracks**		**NavClus**	
	#Recommendations	**#Recommendations per Trace**	**#Recommendations**	**#Recommendations per Trace**
Mylyn	241,084	88	137,973	51
Platform	70,020	120	39,943	69
PDE	12,299	23	6,740	13
ECF	8,836	29	5,312	17
MDT	18,711	76	15,663	64

5.1. Precision, Recall, and F-measure

We present the precision, recall, and F-measure values of the recommendation results of the five projects as well as the average across the five projects in Table 3. The average is weighted by the number of interaction traces held by each project. Table 3 shows that the precision values of NavClus are higher than those of TeamTracks across four of five projects, with the ECF project as the exception. The recall values of NavClus are higher than those of TeamTracks across all of the five projects. The F-measure values of NavClus are higher than those of TeamTracks across all of the five projects. Consequentially, the average precision and recall of NavClus are higher than those of TeamTracks. The average precision and recall of TeamTracks are 0.489 and 0.040 respectively. The average precision and recall of NavClus are 0.522 and 0.082 respectively. Most notably, the average F-measure of NavClus is twice as high as that of TeamTracks (0.073 vs. 0.144).

Table 3. Precision, Recall, and F-measure values for the five projects

Project	TeamTracks			NavClus		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F-measure**
Mylyn	0.449	0.045	0.082	0.508	0.081	0.140
Platform	0.570	0.031	0.058	0.500	0.052	0.122
PDE	0.620	0.029	0.055	0.636	0.115	0.200
ECF	0.348	0.052	0.091	0.383	0.127	0.191
MDT	0.641	0.017	0.034	0.659	0.027	0.051
Average	0.489	0.040	0.073	0.522	0.082	0.144

Fig. 3 shows the F-measure values of NavClus and TeamTracks in bar graph. The F-measure values of projects differ from project to project.The highest one is that of the PDE project, 0.2, while the lowest one is that of the MDT project, 0.05. The relative difference between projects is caused by the numbers of viewed and edited elements in each project (i.e., 15.0 for the PDE project and 58.8 for the MDT project). As we explained in Section 4.4, we measure the recommendation accuracy of ten recommended elements based on the number of viewed elements. Therefore, the PDE project, which has the smallest number of viewed elements (15.0) per interaction trace, yields higher recommendation accuracy than the MDT project, which has a greater number of viewed elements (58.8).

The interesting fact in Fig. 3 is that the gap between the F-measure values of NavClus and TeamTrack is the largest in the PDE project (That of NavClus (0.2) being 3.6 times higher than that of TeamTracks (0.055).) whereas it is the smallest in the MDT project (That of NavClus (0.051) is just 1.5 times higher than that of TeamTracks (0.034)). This implies that the less program elements programmers navigate, the more accurately NavClus recommends program elements than TeamTracks.

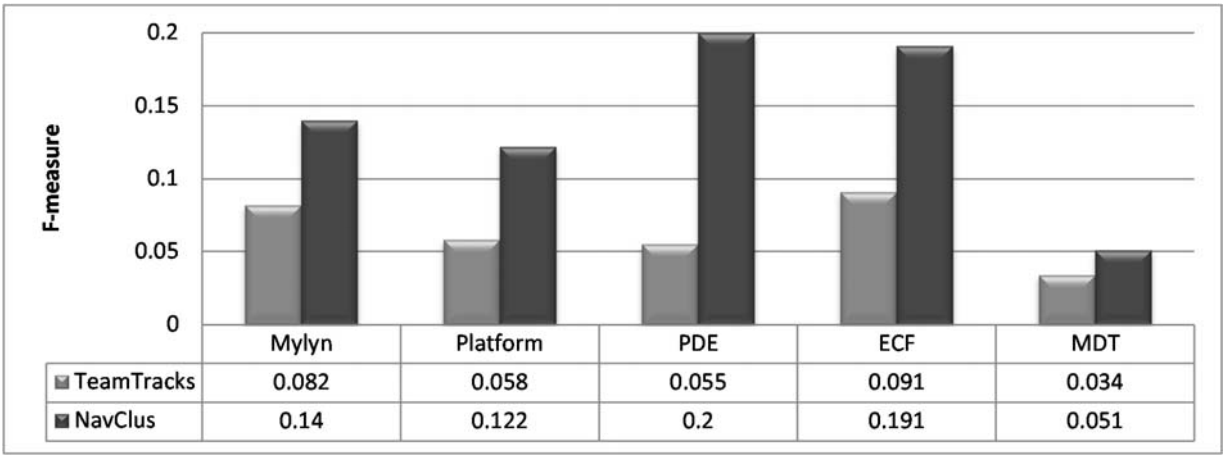

Fig. 3. F-measure values of the five projects

For a more in-depth quantitative analysis, we investigated the recommendation accuracy averaged through the accumulated interaction traces of the Eclipse Platform and PDE projects in Figs. 4 and 5, respectively. In each figure, the X-axis represents the number of accumulated interaction traces and the Y-axis represents the average F-measure of the recommendations that occurred and were accumulated by the number of the interaction traces.

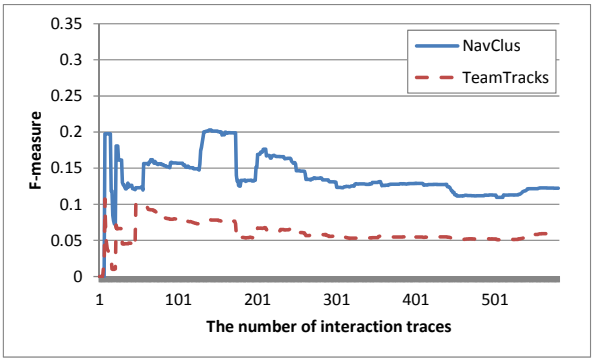

Fig. 4. Average F-measure of the Eclipse Platform project

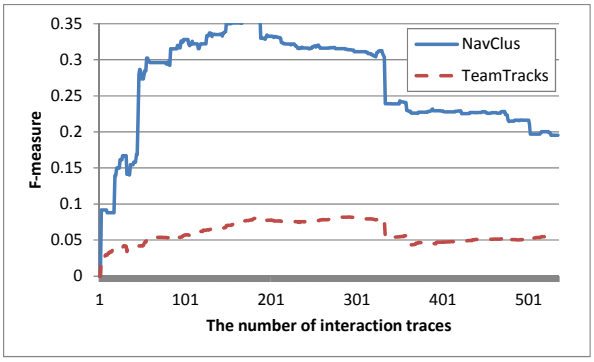

Fig. 5. Average F-measure of the Eclipse PDE project

The key observation through Figs. 4 and 5 is that NavClus maintains higher recommendation accuracy than TeamTracks through interaction traces. The more interesting fact in Figs. 4 and 5 is that the recommendation accuracy does not proportionally increase as interaction traces are accumulated. For example, the Eclipse Platform project shows the highest recommendation accuracy from the 133rd interaction trace to the 155th interaction trace. Similarly, the Eclipse PDE project shows even higher recommendation accuracy (higher than 0.35) from the 150th interaction trace to the 189th interaction trace. This indicates that we may need to consider more principles (e.g. recency) to yield a consistently higher recommendation accuracy through interaction traces.

5.2. Edit-Hit Ratio

The edit-hit ratio represents the ratio of recommendations that include the program elements to edit to the total number of recommendations. Fig. 6 presents the edit-hit ratio values of TeamTracks and NavClus. For example, in the Mylyn project, TeamTracks presents the edit-hit ratio of 0.534: 53.4% of the recommendations of TeamTracks include the program elements that are edited in tasks. In the same project, NavClus presents the edit-hit ratio of 0.733.

Fig. 6 shows that NavClus demonstrates higher edit-hit-ratio values than TeamTracks across four projects, with the MDT project as the exception. The MDT project has an extremely high view-edit ratio (39.2) compared to the other projects (5.4~11.1) as shown in Table 1. However, because the MDT project holds a relatively small number of interaction traces compared to the other projects, the results of the MDT project are less reliable than the other results. Except for the MDT case, NavClus makes useful recommendations more frequently than TeamTracks does.

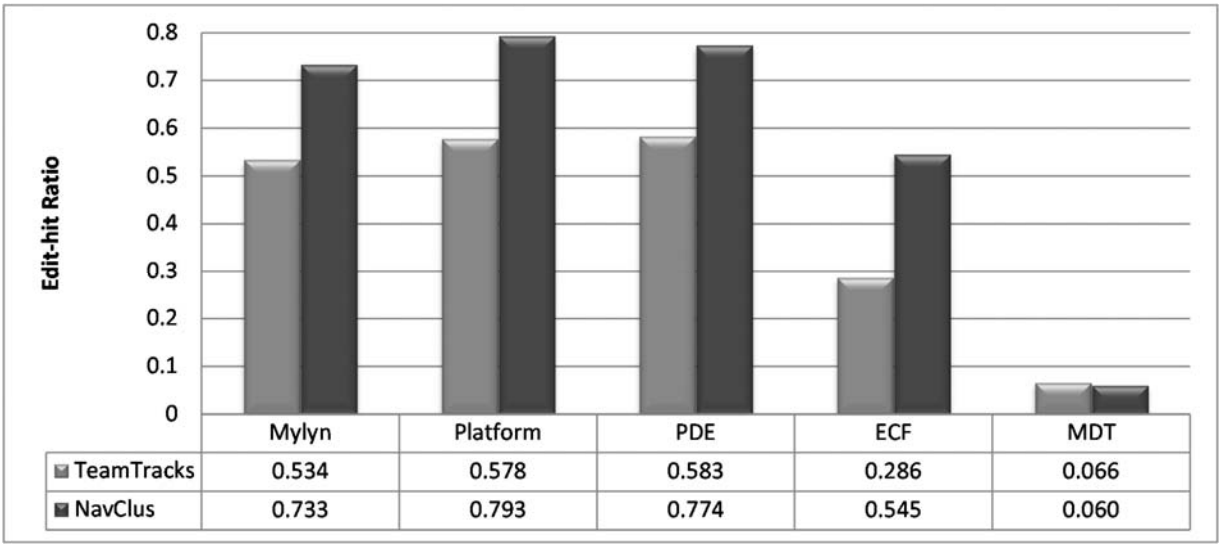

Fig. 6. Edit-hit Ratio values of the five projects

In summary, NavClus yields significantly higher recommendation accuracy than TeamTracks across all five projects. NavClus is designed to collect the program elements that are contextually related, which contributes to capturing a larger contextual scope than TeamTracks, thereby yielding much higher recommendation accuracy than TeamTracks.

5.3. NavClus vs. TeamTracks for Tool Development

We demonstrated in Section 5.1 that NavClus yields twofold higher F-measure values than TeamTracks and in Section 5.2 that NavClus recommends program elements to edit more frequently than TeamTracks. We explain in Section 5.3 how NavClus produces better recommendation results than TeamTracks.

Based upon the improved recommendation results of NavClus, we developed a tool that is equipped with this NavClus approach[7]. The tool is a graphical code recommender which displays recommendations in a UML class diagram to guide a programmer's code navigation by mining upon programmer interaction histories. This was not effective if we used the TeamTracks approach. To explain the reasons, we examine the recommendation results of the Eclipse PDE project, and discuss the recommendation results in this specific tool.

The precision, recall and F-measure of TeamTracks for the PDE project are 0.620, 0.029, and 0.055 respectively. The precision, recall and F-measure of NavClus for the PDE project are 0.636, 0.115, and 0.200 respectively. The recall of TeamTracks is much lower than that of NavClus (0.029 vs. 0.115). As the recall is calculated based on the number of program elements recommended by an approach, we examined how many program elements each approach recommended.

First, TeamTracks recommended four program elements on average at each recommendation for the PDE project. Given precision 0.620, only two program elements were accurate. In addition, TeamTracks recommended one or two program elements in 49.5 % of its 12,299 recommendations. This means that programmers will see one or two program elements in a recommendation. Because this is an insufficient amount of information with a mediocre level of precision, a tool using TeamTracks recommendations may lead programmers astray so that they waste time visiting program elements irrelevant to their tasks.

[7] The tool demo can be found at http://www.youtube.com/watch?v=rbrc5ERyWjQ.

Next, NavClus recommended the maximum number, ten program elements at each recommendation for the PDE project. Given precision 0.636, six out of ten program elements were accurate at each recommendation. This higher level of recommendation accuracy will enable programmers to see more program elements in each recommendation with higher precision, which will allow them to select program elements relevant to their tasks, therefore saving much of their time.

NavClus broadens the contextual scope of the code base that programmers can see as they navigate the code base, i.e. with NavClus programmers can see more program elements that are contextually related than with TeamTracks. As a consequence, NavClus can provide programmers with more precise recommendations than TeamTracks. These differences are crucial factors in making a successful graphical code recommender.

5.4. Threats to Validity

A threat to the validity of this experiment is that these projects used in our experiment may not be representative of all software projects. However, these projects are representative of the class of projects that use the Eclipse Bugzilla system[8]. More specifically, we limit the generalization to projects that have similar view-edit ratios, as our experimental results may depend on the view-edit ratio. Our results are thus generalizable to the open projects that have a view-edit ratio in the range of 5.4~8.6.

Another potential threat is that this experiment may contain information bias because the viewed and edited elements are extracted from Mylyn data only. As we simulated programmers' interactions with the Mylyn data, it might lead to a result that is different from using programmers' actual interactions. However, the difference in the result will be insignificant because the Mylyn data and the actual interactions differ only as follows: Mylyn counts double clicked elements as edited elements and merges the repeated actions on the same program element into one record (Ying and Robillard 2011) but we removed the double-clicked elements from edited elements to avoid counting double clicked elements as edited elements; To keep the time order of the interaction records we sorted the interaction records according to the starting timestamps of the records prior to our simulation. The only parts that we could not recover are the records that merged the same actions on the same program elements, which will not affect the conclusion that NavClus yields higher recommendation accuracy than TeamTracks, however, because NavClus uses both visit frequencies and visit associations while TeamTracks uses only consecutive visits.

6. Related Work

Previous research related to our approach can be divided into two groups: research leveraging programmers' interaction histories (Section 2.1) and research using data clustering techniques (Section 2.2).

6.1. Mining Programmer Interaction Histories

An interaction history is a record of the events in which individuals use computational objects (e.g. documents, menus). Because interaction histories are "a by-product of normal activity and [are] thus essentially free" (Hill et al. 1992), interaction histories can be leveraged to aid programmers at low cost. By utilizing programmer interaction histories, researchers have developed recommendation systems for software maintenance. A recommendation system for software maintenance is defined as "a software application that provides information items estimated to be valuable for a software engineering task in a given context" (Robillard et al. 2010). To recommend program

[8] https://bugs.eclipse.org/bugs/

elements that are relevant to software maintenance tasks, researchers mine programmer interaction histories. This mining research can be classified into association-based mining, frequency-based mining, and others.

1) Association-based Mining

Association-based mining is to mine associations between program elements based on programmer interaction histories. The approaches that are most related to the proposed NavClus in this group are TeamTracks (DeLine et al. 2005) and NavTracks (Singer et al. 2005). NavTracks recommends files to visit next for a programmer currently working on a file (Singer et al. 2005). To recommend such files, NavTracks segments programmer interaction histories into navigation loops and finds associations among files. NavTracks serves as a memory aid to revisit previously viewed files. TeamTracks recommends other methods to visit next, when a programmer visits a method. To recommend such methods, TeamTracks mines consecutive visits between two program elements in programmers' interaction histories (DeLine et al. 2005). In a lab study, TeamTracks, in addition, demonstrated the potential to share navigation data among programmers (DeLine et al. 2005). Likewise, Sahm and Maalej (Sahm et al. 2010) proposed the Switch! approach for recommending subsequent artifacts to visit. This approach makes recommendations when a software developer clicks a button to ask for a recommendation. It recommends various artifacts, including tools and documents. However, the internal mechanism of this approach is similar to that of TeamTracks (DeLine et al. 2005) in that it depends on associations between two artifacts. These approaches show a primitive form of recommending program elements based upon the associations between two program elements, not requiring prior knowledge or explicit input by the programmer. However, these yield low recommendation accuracy. Many of the recommendations are irrelevant or not highly relevant to the given task (DeLine et al. 2005; Lee et al. 2011). To improve the recommendation accuracy, our proposed NavClus clusters navigation sequences to collect program elements that are contextually related to each other. NavClus then counts and utilizes visit frequencies of program elements. The combination of visit frequencies and visit associations contributes to the improved accuracy.

2) Frequency-based Mining

Frequency-based mining is to mine the number of visits on program elements to identify important program elements. The work most related to our research in this group is Mylyn (Kerstenet al. 2006). Mylyncounts the number of interactions that the programmer has with each program element during a task session, calculates the degree of interest of each program element based upon these counts, and shows frequently visited elements. Mylynhelps manage programmers' tasks in the Eclipse IDE. To utilize Mylyn, a programmer identifies a starting time point of a task with a task ID and an ending time point to store a collection of code relevant to the task. The programmer later selects the task ID to restore the collection of code. One shortcoming of Mylyn arises from the fact that Mylyn relies on programmers' manual identifications of tasks. To solve the problem of programmers occasionally neglecting to identify the starting and ending points of tasks and considering that the missed task boundaries may cause program elements to be associated with incorrect relevant tasks, Coman and Sillitti proposed a technique that automatically splits a development session into task-related subsections (Comanet al. 2008). However, while this automation identifies starting and ending points of task sessions, it does not aid subsequent programmers in finding a proper collection. Follow-up studies on helping programmers switch tasks have proposed effective methods for displaying collections of code. First, Safer and Murphy compared an episodic interface (snapshots of a programmer's screen) with semantic interfaces (program elements in a tree view and their interaction events in a timeline) (Safer et al. 2007), and Parnin and DeLine compared the list of a programmer activity in chronological order with the degree-of-interest (DOI) tree view in the style of Mylyn(Parninet al. 2010). Second, Robbes et al. mined the time stamp of events, i.e., the order of changes made by programmers, to reconstruct specific task sessions (Robbeset al. 2007; Robbeset al. 2008). In turn, Hattori et al. proposed the replaying of past changes using the change information in chronological order (Hattori et al. 2010) (Hattori et al. 2011). Third, Rothlisberger et al. proposed displaying the pane in other ways. They proposed HeatMap, which uses an explorer-type view to color frequently and recently visited program elements in red, gradually changing them to blue as time

passes (Rothlisbergeret al. 2009), Autumn Leaves manages the window tabs opened by a programmer and automatically closes or grays out tabs that are unlikely to be used in the future (Rothlisbergeret al. 2009), and SmartGroup manages task-relevant source artifacts, similarly to Mylyn, but automatically identifies a task type i.e., defect-correction and/or feature-implementation tasks (Rothlisbergeret al. 2011). These approaches typically require the task boundaries to be manually identified by a programmer and the programmer's explicit trigger to present program elements relevant to tasks. In addition, to restore a collection of code, programmers still need to manually indicate a task ID. This requires prior knowledge and explicit effort by the programmers. Our approach differs from this group of approaches in that it does not deal with task boundaries. This enables programmers to receive collections of code relevant to tasks without prior knowledge or explicit effort, and thus makes it possible that collections of code relevant to tasks are shared automatically.

3) Other Mining

Researchers have conducted empirical studies to find new facts from programmer interaction histories as well. For example, Parnin et al. explored several characteristics of mining programmer interaction histories and revealed several important findings, including the finding that "discarding the least recently used method" helped recover contexts (Parnin et al. 2006). Zou, Godfrey and Hassan classified "interaction couplings," the association between program elements, which can be extracted from programmer interaction histories into co-change, view and change, and co-view relationships (Zou et al. 2007). Fritz et al. proposed a degree-of-knowledge model that is formed from programmer interaction histories to determine which programmer has expertise on which part(s) of the code base (Fritz et al. 2010). Ying and Robillard analyzed programmer interaction histories, finding that programmers view program elements for a while and edit program elements finally in an enhancement task, while programmers edit program elements relatively soon in a bug fix task (Ying et al. 2011). The remaining research on programmer interaction histories developed new metrics using programmer interaction histories. Robbes et al. proposed a benchmark to evaluate the accuracy of change predictions by considering the order of the changes (Robbes et al. 2010). Lee et al. identified 56 metrics in programmer interaction histories, including time properties, and built a classification model based upon them to predict when a file will have a defect (Lee et al. 2011). However, these empirical studies focused on finding new patterns in programmer interaction histories, not directly proposing a new technique for a recommendation system. Our approach is a new technique to recommend collections of code for effectively guiding code navigation.

We suggest a clustering technique that automatically shares collections of code relevant to tasks. In our previous work in (Lee et al. 2011), we demonstrated a case study of NavClus. However, at that time the NavClus algorithm had not been fully developed. The case study was limited to a small amount of experimental data that were logged while twelve programmers were repeatedly performing the same four tasks. In this proposed work, we develop and present the NavClus algorithm. We also utilize the NavClus algorithm by inputting real massive data that were logged when programmers performed practices. Our technique yields a higher recommendation accuracy than the association-based mining approaches. This technique differs from the frequency-based approaches in that it does not deal with task boundaries, thus not requiring programmers' prior knowledge and explicit effort. Our work is an advance of the previous work that implicitly recommends program elements to guide code navigation.

6.2. Data Clustering Techniques

Data clustering techniques are employed to group similar elements into clusters with the goal of obtaining high intra-cluster cohesion and low inter-cluster coupling. Depending on the starting point of grouping elements, clustering techniques can be classified into four classes: divisive clustering (e.g., k-means (Han et al. 2000)), agglomerative clustering (e.g., Birch (Han et al. 2000) and CluStream (Aggarwal et al. 2003)), density-based clustering (e.g., DBSCAN (Han et al. 2000) and TraClus (Lee et al. 2007)), and grid-based clustering (e.g., STING (Han et al. 2000)).

Robillard and Dagenais propose retrieving collections of code through software revision histories (Robillard et al. 2010). They used a nearest-neighbor clustering algorithm that depends on k number of overlapped elements through change sets. However, the resulting clusters are the common parts of change sets, and thus are not related to particular tasks. The k-nearest neighbor clustering technique produces clusters of a very low quality that did not meet our expectations for recommended collections of code relevant to tasks. To produce better clusters, we propose our clustering technique. The proposed technique differs from the k-nearest neighbor clustering technique not only in that the proposed technique mines different information source, programmer interaction histories, but also in that it proposes splitting points and merging points of programmer interaction histories and offers a clustering model to produce collections of program elements that may be relevant to the tasks performed by programmers.

The data clustering techniques that inspired us are CluStream (Aggarwal et al. 2003) and TraClus (Lee et al. 2007). To mine a data stream efficiently, CluStream (Aggarwal et al. 2003) clusters evolving data streams in two phases, micro-clustering and macro-clustering. Micro-clustering is carried out to generate a summary (a micro-cluster) from an incoming data stream online, and macro-clustering serves to cluster micro-clusters offline. Our approach employs this type of two-phase clustering. TraClus (Lee et al. 2007) partitions a trajectory into a set of line segments, after which it groups line segments that are similar in order to find common sub-trajectories. Our approach uses this partition-and-group clustering concept[9]. However, we do not use the details of these two techniques, as these approaches deal with numeric/point/position data and are thus difficult to apply to programmer interaction histories, which are non-numeric data streams. Thus, we propose a new clustering method that deals with programmer interaction histories that contain non-numeric data.

7. Conclusion

We developed a technique that clusters navigation sequences from programmer interaction histories to automatically mine collections of code that are relevant to tasks. We also proposed the NavClus approach that adopts this clustering technique to recommend collections of code that are relevant to a task at hand. We evaluated NavClus by simulating code recommendations, measuring the recommendation accuracy, and comparing the recommendation accuracy with that of the state-of-the-art approach TeamTracks. The evaluation demonstrated that NavClus provides significantly higher recommendation accuracy than TeamTracks. The key insight which enabled achieving such a high level of recommendation accuracy is that mining collections of program elements that are contextually related can provide a basis for recommending program elements that are highly relevant to a given task.

In the future, we will continue to research and develop a context-aware code recommender that recognizes a programmer's situation and uses this information to provide program elements relevant to the programmer's task at hand, without requiring programmers' explicit effort or prior knowledge. We plan to extend this work as follows:

1) We will investigate whether the collections of code clustered by our approach may be related to features. Our conjecture is that clusters mined in programmer interaction histories may reflect on collections of program elements of which a feature is composed. We will verify this conjecture.

[9] Our approach segments programmer interaction histories into sets of navigation sequences and then groups similar navigation sequences.

2) We will compare different techniques for mining programmer interaction histories. For example, we will compare Hidden Markov Model (HMM) with our approach. We believe this comparative result will contribute to the creation of more accurate recommendations.

3) We will conduct user studies in which programmers work with NavClus. In particular, we believe we can reduce the time required for new team members to understand the code base. To determine the usefulness, benefits and difficulties of NavClus, we plan to conduct a series of user studies.

8. Acknowledgements

We thank to Sunghun Kim for providing the data used in the experiment, and to anonymous reviewers for providing valuable comments. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0007069).

9. References

Aggarwal, C.C., Han, J., Wang, J., and Yu, P.S., 2003. A framework for clustering evolving data streams. In Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB '2003), Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.), Vol. 29. VLDB Endowment 81-92.

Coman I. and Sillitti, A., 2009 Automated segmentation of development sessions into task-related subsections, International Journal of Computers and Applications, vol. 31, pp. 159–166.

Dey A., 2001, Understanding and using context. Personal Ubiquitous Computer. 5(1), Jan. 2001, pp. 4-7.

DeLine, R., Czerwinski, M. and Robertson, G., 2005, Easing program comprehension by sharing navigation data, In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '05). IEEE Computer Society, Washington, DC, USA, pp. 241-248.

Fritz, T., Ou, J., Murphy, G. C. and Murphy-Hill, E., 2010. A degree-of-knowledge model to capture source code familiarity. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10), Vol. 1. ACM, New York, NY, USA, pp. 385-394.

Han J. and Kamber, M., 2000. Data Mining: Concepts and Techniques, Morgan Kaufmann.

Hattori, L., Lungu, M., and Lanza, M. 2010. Replaying past changes on multi-developer projects. In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) (IWPSE-EVOL '10). ACM, New York, NY, USA, pp. 13-22.

Hattori, L., D' Ambros, M., Lanza, M., Lungu, M. 2011. Software evolution comprehension: replay to the rescue. Program Comprehension (ICPC), 2011 IEEE 19th International Conference on , vol., no., pp.161-170.

Hill, W. C., Hollan, J. D., Wroblewski, D. and McCandless, T. 1992. Edit wear and read wear, Proceedings of the SIGCHI conference on Human factors in computing systems, Monterey, California, United States, pp. 3-9.

M. Kersten and G.C. Murphy, 2006. Using task context to improve programmer productivity, In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14). ACM, New York, NY, USA, pp. 1-11.

Ko, A.J., Aung, H.H. and Myers, B.A. 2005. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks, In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, pp. 126-135.

Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung. H. H., 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. Software Engineering, IEEE Transactions on , vol.32, no.12, pp.971-987.

LaToza T.D. and Myers, B.A., 2010 In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10), Vol. 1. ACM, New York, NY, USA, pp. 185-194.

Lee, J.G., Han, J., and Whang. K.Y., 2007. Trajectory clustering: a partition-and-group framework. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07). ACM, New York, NY, USA, pp. 593-604.

Lee, S. and Kang, S,. 2011. A study on guiding programmers' code navigation with a graphical code recommender, Studies in Computational Intelligence, Vol. 377, Springer, pp. 61-75.

Lee, S. and Kang, S,. 2011, Clustering and recommending collections of code relevant to tasks, Software Maintenance (ICSM), 2011 27th IEEE International Conference on , vol., no., pp.536-539.

Lee, T., Nam, J., Han, D., Kim, S. and In, H. P. 2011. Micro interaction metrics for defect prediction. In Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011). Pp. 311-321.

Mark, G., Gonzalez, V.M. and Harris. J. 2005. No task left behind? Examining the nature of fragmented work. In CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, 2005. ACM Press, pp. 321–330.

Masud, M., Chen, Q., Khan, L., Aggarwal, C., Gao, J., Han, J., and Thuraisingham, B., 2010. Addressing concept-evolution in concept-drifting data streams In Proc. of 2010 10th IEEE International Conference on Data Mining (ICDM 2010), Sydney, Australia, pp.929-934.

Parnin, C. and Görg, C. 2006. Building usage contexts during program comprehension. In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06). IEEE Computer Society, Washington, DC, USA, pp. 13-22.

Parnin, C. and DeLine, R. 2010. Evaluating cues for resuming interrupted programming tasks. In Proceedings of the 28th international conference on Human factors in computing systems (CHI '10). ACM, New York, NY, USA, 93-102. pp. 93-102.

Robbes, R. and Lanza, M. 2007. Characterizing and understanding development Sessions. Program Comprehension (ICPC '07). 15th IEEE International Conference on , pp. 155-166.

Robbes, R., Pollet, D., and Lanza, M. 2008. Logical coupling based on fine-grained change information. Reverse Engineering, WCRE '08. 15th Working Conference on, IEEE CS Press, pp. 42 - 46.

Robbes, R., Pollet, D. and Lanza, M. 2010. Replaying IDE interactions to evaluate and improve change prediction approaches. In Proceedings of the Working Conference on Mining Software Repositories. pp. 161-170.

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

Robillard, M., Walker, R. and Zimmermann, T. 2010 Recommendation systems for software engineering. IEEE Software. pp. 80-86.

Rothlisberger, D., Nierstrasz, O., Ducasse, S., Pollet, D. and Robbes, R. 2009. Supporting task-oriented navigation in IDEs with configurable HeatMaps. In Proceedings of the IEEE 17th International Conference on Program Comprehension, ICPC'09. pp. 253–257.

Roethlisberger, D., Nierstrasz, O., Ducasse, S., 2009. Autumn Leaves: curing the window plague in IDEs, 16th Working Conference on Reverse Engineering, pp.237-246.

Rothlisberger, D., Nierstrasz, O., Ducasse, S., 2011. SmartGroups: Focusing on Task-Relevant Source Artifacts in IDEs, Program Comprehension (ICPC), 2011 IEEE 19th International Conference on , pp. 61 – 70.

I. Safer and G.C. Murphy, 2007. Comparing episodic and semantic interfaces for task boundary identification, In Proceedings of the 2007 conference of the center for advanced studies on Collaborative research (CASCON '07). ACM, New York, NY, USA, pp. 229-243.

Sahm, A., Maalej. W., 2010. Switch! Recommending Artifacts Needed Next Based on Personal and Shared Context. In Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner, editors, Software Engineering 2010 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26.02.2010, Paderborn. Volume 160 of LNI, pp. 473-484.

Singer, J., Elves, R., and Storey. M.-A., 2005. NavTracks: supporting navigation in software maintenance. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05). IEEE Computer Society, Washington, DC, USA, pp. 325-334.

Wang, J., Peng, X. Xing, Z. Zhao, W., 2011. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions, Software Maintenance (ICSM), 2011 27th IEEE International Conference on , vol., no., pp.213-222.

Wilde N., Gomez J.A., Gust T. and Strasburg, D. 1992. Locating user functionality in old code, In Proceedings of International Conference on Software Maintenance, pp. 200-205.

Woods S. and Yang. Q. 1996. The program understanding problem: analysis and a heuristic approach, In Proceedings of the 18th international conference on Software engineering (ICSE '96). IEEE Computer Society, Washington, DC, USA, pp. 6-15.

Ying, A. T. T., and Robillard, M. P. 2011. The influence of the task on programmer behavior. In Proceedings of the 19th IEEE International Conference on Program Comprehension, pp. 31-40.

Zimmermann, T., Weissgerber, P., Diehl, S. and Zeller, A. 2005. Mining version histories to guide software changes. IEEE Transactions on Software Engineering. 31(6), pp. 429-445.

Zou, L., Godfrey, M.W. and Hassan, A.E. 2007. Detecting interaction coupling from task interaction histories. In Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07). IEEE Computer Society, Washington, DC, USA, pp. 135-144.