Building Usage Contexts During Program Comprehension

Chris Parnin, Carsten Görg*
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
{vector,goerg}@cc.gatech.edu

Abstract

Software developers often work on multiple simultaneous projects. Even when only a single project is underway, everyday distractions interrupt the development effort. Consequently, developers spend significant effort pursuing recovery of their context. By context, we focus on the classes and methods within the code that are relevant to a specific bug being fixed or enhancement made. Context is reified by a program in terms of a set of presentations (windows a containing source code, command executions, and data files); however, it is not enough to save the latest context. Even when working on a single task, programmers flip between contexts as they extend their understanding, and when they decide on a change, they may have to visit several contexts in order to address all possible ripple effects. Consequently, we would like to record a history of contexts and be able to retrieve them as demanded by the current task. We introduce a novel technique to obtain a context, consisting of a set of methods relevant for the current task, from a programmer's interactions with an IDE. Using this context, we demonstrate how to improve the ability of a programmer to recover the mental state associated with tasks and to facilitate the exploration of software through recommendation systems.

1. Introduction

The professional programmer must bear mental burdens from performing a complex task in an environment constantly contending for attention. In a typical daily routine, a programmer can be expected to attend a meeting, read email, converse with teammates, go to lunch, be interrupted with a higher priority problem, advise a colleague who dropped by, and resolve personal issues. These activities and distractions detract from the mental resources needed to perform programming tasks requiring considerable mental energy (*e.g.* program comprehension).

Professional programmers take several steps to alleviate these problems. In interviews and observations with programmers, we noted several techniques used. One user transcribed his activities and related artifacts in a word-processing application. Another used an email program's task list to maintain a to-do list and bookmarks in his programming IDE to maintain his current location. Another programmer wrote comments in code to mark issues and unfinished work. These observations are similar to those observed in case studies of information workers [6, 9].

Several researchers have shown that information workers are plagued with problems arising from environmental distractions occurring in the work place. One such problem, *prospective memory failure*, involves a failure to recall the need to perform a task at the appropriate time. Interruptions during the performance of a task have been cited as a primary cause of prospective memory failure. Interruptions also have negative effects on a task's performance. In an *in situ* diary study, it was found that tasks which are resumed from an interruption were rated more difficult to perform, took twice as long, and thus were more likely to experience further interruptions [3]. Another study showed that when a task is interrupted, 40% of the time the task is not resumed immediately [12].

When programmers perform tasks on complex systems, understanding the system in its entirety often becomes an insurmountable task. Instead, programmers often distill the complex system into the essential elements needed to accomplish their task. These elements can be methods, classes or files in the systems. The process of finding and understanding these essential elements is called *software exploration* (search). Studies of the work practice of programmers [14, 17] confirm search as the prevalent task per-

* The author was supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD).

formed by programmers. Zayour *et al.* [17] describes the cognitive difficulties imposed on the user by searching: they "have to recall their plans and models upon returning to an earlier context", and "jumping window to window can be time consuming and it requires mental energy to remember where everything is located".

In our work, we capture the sections of the source code the programmer is currently interested in for the purpose of assisting the programmer in (1) recovering mental state and (2) facilitating the exploration of the source code. In particular, we have explored using a user's previous interactions with a programming IDE to obtain a *context*, a set of relevant methods with respect to a task or point in time. Using context, we demonstrate the construction of a recommendation system that presents relevant suggestions while balancing the need to recover relevant entities.
The contributions in this paper are as follows:

- a set of models and techniques for obtaining a context through data mining of interaction history,
- the algorithms for an application of a context-recommendation system which balances the ability to recover previously accessed methods while presenting new methods for exploration,
- a case study of professional programmers demonstrating the efficacy of our technique for improving *recovery* and *exploration* in comparison to standard recommendation systems.

In the following sections, we relate our work to previous research efforts, propose some potential applications applying the concept of context, develop an interaction history event model, describe methods for obtaining a context from interaction history, describe the design of an application of a context-recommendation system, and finally present a case study where we examine our approach on interaction history obtained from professional programmers.

2. Related Work

Several tools and methodologies have been proposed to mitigate the cognitive load incurred during exploration. To reduce short term memory (STM) overload during tool usage, Zayour and Lethbridge [17] suggested reducing the number of artifacts in STM by "achieving visual proximity between related artifacts", by "facilitating meaningful encodings" for chunking information, and by "reducing the uncertainty during exploration". To reduce the time artifacts remain in STM, they suggested minimizing the time and complexity spent acquiring a related artifact. Storey *et al.* defined an even more comprehensive set of cognitive design elements to consider for software exploration tools by improving program comprehension and reducing the maintainer's cognitive overhead [15].

While these approaches have been successful in the development of code browsers and software visualization systems, not all of these approaches are applicable with recommendation systems, sometimes referred to as remembrance agents. The interfaces of recommendation systems are much more light-weight than code browsers, often comprising of just a textual list of recommendations.

In software engineering, several recommendation systems have attempted to improve program understanding and navigation. Ye and Fischer [16] mined revision history and used remembrance agents to prevent code duplication by identifying methods which can potentially be reused. In `eROSE` [18], Zimmermann *et al.* used association rules to offer suggestions about related changes when a user was editing a method using the `ECLIPSE` IDE. Their goal was to facilitate impact analysis by informing the user of changes associated with a proposed change. However, mining from revision history has some limitations: it does not necessarily capture fine-grained interactions and misses the interactions of users browsing and searching source code.

Other researchers have developed recommendation systems which analyze navigation history in order to recommend possible locations of interest. In `NavTracks` [13], navigation loops are recovered from recent navigation paths and the files related to the current method are displayed. In `FAN` [4], navigational history is analyzed to display a list of methods that are accessed next after visiting the current method. In addition, `FAN` used the frequency of navigations to indicate the degree-of-interest (DOI) of elements in an UML diagram. The DOI model used in `FAN` provides a global view of the source code; this approach has difficulty in assisting programmers working on elements not historically having high DOI. In `MYLAR` [10], the DOI model is calculated from keystrokes and selection of elements. `MYLAR` differs from `FAN` in decaying the DOI over time; in effect, creating a DOI model relevant to the current task. With `MYLAR`, the DOI model can be adjusted to be relevant with respect to the task occurring over the past few days; however, the approach is not suitable for deriving a context. The high frequency of past elements creates a barrier to entry for recently visited elements and hence it cannot adapt to programmers transitioning between sub-contexts.

2.1. Our Approach

In our work, we consider how to obtain a set of related methods for any given moment in time, not just the current method. In most recommendation systems, items are only relevant with respect to a current method. This makes exploration difficult because previous recommendations are replaced when following the current recommendation to the new location.

In this paper, we are focused on analyzing how programmers interact with source code and using these insights to develop techniques that support recovery and exploration. We argue that our approach is more suitable for facilitating exploration and maintaining a mental context while performing programming tasks. Instead of displaying a list of recommended methods, we maintain a set of relevant methods and gradually introduce recommendations.

Although previous systems have demonstrated success in facilitating navigation, we improve on these using the simple approach of maintaining a context of recently accessed methods. Furthermore, we can handle data sets that are much larger than many of the programs evaluated in previous studies.

3. Usage Contexts

In this section, we discuss applications of contexts derived from a programmer's interaction history to address several software comprehension problems.

At a given point in time, a programmer can be interested in a subset of elements of a program. An *element* may be a method, class, or file. This subset of elements is called a *context*.

In this paper, we examine when programmers interact with a method and use this to derive an explicit representation of context. We choose methods as the program element of interest as a middle ground between details of statements and abstractions provided by classes; representation of the context may choose to display a "snippet" of the statements in a method.

3.1. Recommendation System

A context can be presented as a list of recommendations to the programmer. The representation can be a simple textual list of methods or a light-weight peripheral visualization tool run on a second monitor. The elements of the context would serve as a way for the programmer to reinstate their mental context after a period of distraction, or reducing the cognitive load and disorientated associated with performing a search. Further, the context can be used as a platform for inserting recommendations from other tools. The advantage of this approach is that (1) the recommendation system can use the context as a basis for implicit query into the recommendation dataset, (2) the lifetime of how long recommendations are accessible is extended, and (3) the cost of following recommendations is reduced because previously accessed methods are still recoverable.

3.2. Context Filters

Contexts can be used as a DOI filter in both the IDE and other software tools. Consider the following scenarios:

It has recently been discovered that bug fix 273 has not actually been fixed; however, the developer who performed the fix has long since left. Instead of painstakingly searching the code base for all related methods to understand where the developer left off, an enterprising intern loads the context formed by the developer performing the bug fix that day. This context now serves as a filter in the intern's IDE to facilitate understanding the code. This differs from revision history because the context also contains methods visited by the developer.

A programmer has spent all day debugging a very large program, but cannot seem to locate the source of the problem. The programmer decides to launch `Tarantula` *[8], a fault visualization tool. Often,* `Tarantula` *provides too many fault locations for this programmer to investigate because there are currently many bugs in the system. Since the context of the programmer is accessible,* `Tarantula` *can load the context of the programmer and use it to filter unrelated information.*

Some possible applications of context can be used in the following manner:

- to recover the mental state of a programmer after disorientation,
- as filters in software engineering tools,
- in recommendation systems to recover mental state and facilitate exploration,
- to recover the methods of past tasks.

4. Interaction History

In this section we provide background information about interaction history and describe our model for abstracting interaction events.

4.1. Background

Interaction history is a record of a user's interactions with an application for the purpose of providing insight into that history as well as facilitating future interactions. Alternative terms for interaction history include *navigation history*, *user history*, *computational wear*, *edit wear*, *source code wear*. The first discussion of interaction history emerged from work on *edit/read wear* [7]. *Wear* is the concept of digital objects embedding the history of interactions, much like the dog-eared pages in a book indicate favorite passages. As an example, a text document records how often a line was edited. The frequency of editing a line is then conveyed in a line-based visualization that is embedded in the scrollbar of the document.

4.2. Interaction Event Model

Applications provide a diverse set of interactions made available to a user. We want to abstract the interactions a programmer makes with an IDE in order to reason about the semantic implications. A model that takes the application state into account while explaining the consequences and dependencies of interactions requires considerable effort; there are several candidate parameters to draw from: properties of the user, application, data, and interactions.

The categories of interactions with an IDE are as follows:

navigation: A command used to go to a location in a file,

click: A mouse selection of a method within a code editor,

edit: A change in a line of code,

query: A search for the location of a method.

5. Obtaining Context from Interaction History

What information does an interaction history of a programmer convey? An interaction history catalogs the time and name of methods a programmer has interacted with: in other words, interaction history provides insight into which methods a programmer has expressed interest during a *session* of activity. A session is a sequence of interaction events that have occurred within a given window of time. Choosing an appropriate window of time is highly dependent on the application, in this paper; we assume a session is one day of activity.
Consider the following interaction event stream:

$$\boxed{\text{AAABABAAAAAXYACCDDADABAACDBA}}$$

The most straightforward approach for obtaining a context is to simply collect all methods that were interacted within a given session. The resulting context is:

$$\{A,B,C,D,X,Y\}$$

Note that some of the methods appear more frequently than others; further, some methods appear more often consecutively than others (*e.g.* A compared to B). Some of the methods are sparse and appear isolated (*e.g.* X,Y).

By using this approach, we have obtained a set of relevant methods – relevant because the user expressed interest in the method; however, we may not have obtained a useful context. For our application of supporting recovery of mental state and exploration, a smaller context is more desirable.

Contexts derived from larger sessions are likely to contain too many methods; therefore, irrelevant methods must be filtered out of the interaction history. Of the methods a programmer has interacted with, what measures of "interesting" can be applied? By examining the structure of interaction history, we can gain insight into how to extract the relevant methods. In particular, there are two access patterns to investigate: how long a programmer stays in a method, and the transition patterns between methods. A *transition* is a change of location to another method.

5.1. Localized Frequency

The amount of time or frequency with which a programmer interacts with a method can be used to indicate interest. Once a method is selected, a programmer may further interact with the method by clicking within the method (common with longer methods), highlighting code, and performing edits. This continues until the programmer transitions to another method. Time can be measured in seconds or using the sequence number of the event; for this paper, we use the sequence number as a unit of time.

We develop two models of programmer interactions for understanding the activities of programmers.

Intensity. The intensity of an interaction event at a point in time is *the number of prior consecutive interactions of the same method during the session.*

For the event stream "AAABBC", the respective intensity of each interaction is 0,1,2,0,1,0.

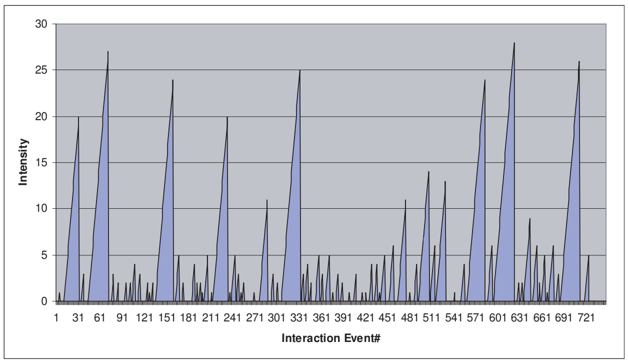

Figure 1. A graph of the event's intensity.

Although a programmer may interact with many methods in a course of a session, only few exhibit high intensity. One possible interpretation of intensity is that method groups with high intensity are the targeted methods of interest to a programmer. In the *valleys* between two peaks of high intensity are smaller peaks of medium and low intensity activity. These lower intensity activities can result from the need to correct compile errors, update references, and search for the next item of interest. This trend can be seen in a visualization of the programmer's activity in Figure 1.

Intensity takes a simplified discrete view of localized frequency; in reality, a programmer may need to briefly transition away from a method, and then return back. This requires a continuous evaluation of localized frequency which we call *momentum*.

Momentum. The momentum at time t_n of an interaction event is:

$$momentum(t_n) = intensity(t_0) * e^{-rt_n}$$

where r is the *discount rate* which regulates the speed of exponential decay, t_0 is the time the event was initiated, and t_n is n time units after t_0.

Instead of having a value of zero after a transition, momentum decays exponentially in intensity. An interaction event having an intensity of 20, with a discount rate of 0.1 has a momentum of 7.35 after 10 steps and a momentum of 0.99 after 30 steps. There is a small twist; a method that is decaying can be reinvigorated if the method is revisited. In this case, the remaining momentum accumulates with the newer intensity, and t_0 is reset to be the new time.

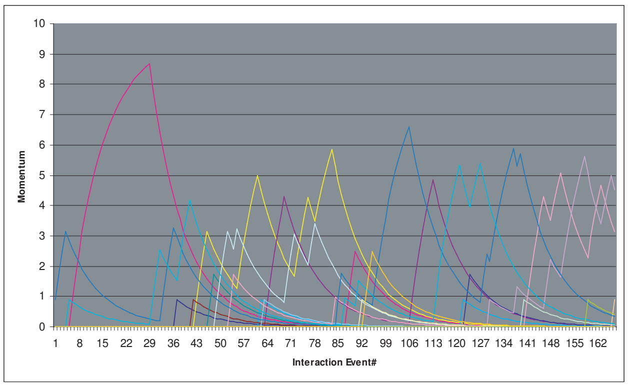

Figure 2. A graph of the event's momentum.

Momentum gives a better indication of which methods are active during a window of time as seen in Figure 2; however, its continuous nature can make it more difficult to use in some applications.

Intensity is useful for filtering out interactions. When users navigate source code, they often quickly "thumb" through the code to locate the next method they are interested in. We decided to filter out any interaction with intensity zero to discard "navigational jitter" [13], a rapid period of transitions between methods often occurring during searching.

5.2. Transitions

A *transition* is a change of location to another method. Transitions are useful in reasoning about a stream of events because they provide an atomic unit of navigation and can be used to categorize events. The programmer is considered to have a current location – the method the programmer is interacting with at that moment. Much like a state machine, the programmer can remain in this current method or transition to another method. We identify several transition types:

recovery transition: A transition to a location previously visited in a session,

exploration transition: A transition to a location not previously visited,

intra-class transition: A transition within a class,

inter-class transition: A transition to a method in another class.

Figure 3 gives an example of these categories of transitions.

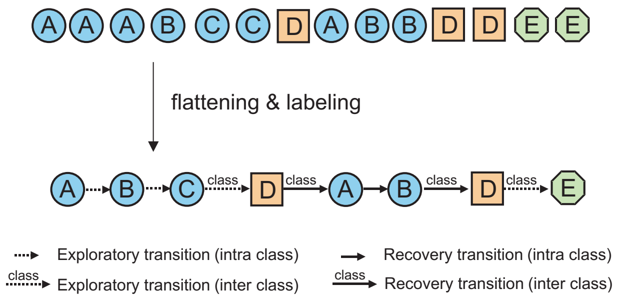
flattening & labeling
class
Exploratory transition (intra class)
Exploratory transition (inter class)
Recovery transition (intra class)
Recovery transition (inter class)

Figure 3. Example for labeling transitions.

6. A Context-Recommendation System

A common interface technique in recommendation systems is to provide a list of *k* items related to an active entity. The size of *k* is restricted by the available screen real estate, and, to a smaller extent, the precision and recall of related items. The assumption behind the recommendation list is that accessing the entity from the list is *cheaper* in time or mental energy than accessing it through other available interface mechanisms.

A similar approach can be used to construct an example application using context – displaying the active methods using a list of *k* items in the IDE. The goal is to effectively support recovery and exploration by maintaining the most relevant methods in the context list.

The advantage of this approach is that instead of being directly concerned with discovering the most important methods, the problem becomes a replacement problem; the least relevant method is discarded. This scheme supports the recovery of previously accessed methods well – the programmer can explore code, or be interrupted by an environmental distraction and upon returning to the task, access the most relevant methods at that point of time before interruption.

Although, recovery of previous methods seems possible, how can a context-recommendation system also suggest other related methods to a programmer? The recommendation system could introduce new suggestions into the context. In the interface, recommendations are clearly distinguishable from previously viewed methods. There are benefits of this approach for the programmer: (1) the *lifetime* of the recommendation is extended and, (2) the *cost* of following a recommendation is reduced. For (1), recommendations are maintained in the context until they are no longer relevant; this is in contrast with other approaches where the recommendation list clears if the programmer decides to visit another method. Longer access to recommendations increases the usefulness of the recommendation by allowing the programmer to defer when the recommendation is followed. For (2), a programmer can follow the recommendation without worrying about losing other recommendations in the list, and can easily return to previous methods.

Often users are only genuinely interested in a small percentage of recommendations given to them. The value of recommendations can be increased by balancing how many recommendations are introduced against the need to recover previously accessed methods.

6.1. Replacement Algorithms

When no more available slots remain in the context, the least relevant method must be discarded. The following are different replacement algorithms; the insight behind the algorithms share much in common with page caching algorithms used to reduce page faults: either the localized frequency or the age of methods is exploited.

LFU: Discard the method *least frequently* used,

MOMO: Discard the method with the *least momentum*,

FIFO: Discard the *oldest* method,

LRU: Discard the method *least recently* used,

OPT: Discard the method that will not used for the longest period of time in the future.

The LFU (least frequently used) algorithm replaces the method with the lowest number of interactions. This approach can be successful in discarding irrelevant methods; however, if the usage period is not evenly distributed, it could cause the method to stay in the context too long. MOMO (momentum-out) is a variation of this algorithm where the frequency is exponential decayed.

The FIFO (first-in first-out) algorithm performs well if a method is encountered early, and not used later on; however, it may perform poorly on methods recurring throughout the duration of a task. LRU (least recently used) algorithm replaces the method that has not be used for the longest period of time. Belady's OPT (optimal) algorithm [2] provides the best possible replacement to minimize total miss rate. The algorithm examines the future stream to evict the method that will not be used for the longest period of time.

6.2. Prefetching Algorithms

Without further enhancement, the contents of the context is limited to recovering methods previously encountered – the context will always miss on methods never previously encountered. Recommendations must be "prefetched" into the context.

Any recommendation system would be suitable; however, we select two recommendation systems `NEXT` and `ASSOC` to evaluate in this paper. The first system suggests methods related to the current context by selecting methods most typically accessed next after the current method. The second system suggests methods associated with the context by using association rules mined from interaction history.

The first system is actually an implementation of a recommendation system described in `FAN` [4], a system which suggests methods which are frequently accessed next after viewing the current method.

Association rules describe associations between methods by calculating the co-occurrence of methods across all sessions. Association rules can yield a recommendation for a set of methods. Association rules can be found by calculating methods that frequently occur together. Frequent itemsets are generated by the Apriori algorithm [1] and then trimmed by requiring a minimum criteria in the statistical significance of the result. Two common measurements are *support*, the number of examples an itemset is present in, and *confidence*, the percentage of time the recommendation is correct in the example itemsets.

Although association rules have not been applied to the interaction history of programmers, many similar systems exists. Several researchers have used association rules in recommendations for web navigation [11, 5] and in `eROSE` [18]. In `eROSE`, revision history was mined to suggest methods that are likely to be changed when proposing a change to a method.

7. Case Study

We conducted an exploratory case study to obtain interaction history from professional programmers; in particular, we sought to evaluate the following questions:

1. What are the properties of interaction history?
2. How much recovery does a context support?
3. How effective are recommendations from interaction history?
4. How effectively does context support both recovery and recommendations?

7.1. Interaction History of Visual Studio

Ten employees from a defense contractor volunteered to participate in this study. The projects the employees worked on were written in C++ and C# and varied in size from 50K to 200K lines of code. Some projects were over 10 years old while others were in initial development.

We created a Visual Studio plug-in called `InteractionHistoryDB` that recorded the interaction history of programmers for 30 working days. The plug-in registered events exposed through the Visual Studio add-in interface and logged the active method targeted by the interaction.

In Section 4.2, we identified four interaction types, however, we only recorded click, navigation, and edit interactions in our experiments. Click events were recorded using a mouse message hook. Navigation events included using commands such as Goto Definition, changing the active edit tab, selecting a class or file and, navigating from a Find-In-Files result. An edit action was recorded by listening to an event raised when a line of code is changed. The edit event is not raised until after the user changes focus from the line being edited. We ignored scrolling and page-up and page-down events because they did not strongly indicate interest in a particular method.

Visual Studio includes direct manipulation tools such as an interface editor. However, to simplify the scope of this study, we choose to ignore the interaction types associated with direct manipulation interfaces.

7.2. Attention and Access Patterns

In a series of experiments, we analyzed the interaction history of programmers to gain insight into the nature of the activities professional programmers performed during a typical work day. In particular, we looked at the following questions.

How many methods did programmers work with in a day?

Understanding the distribution of the number of methods accessed in a day helps demonstrate how much of a cognitive load is imposed on a programmer and provides an assessment of the maximum size of a context.

For each day, we counted how many methods a programmer interacted with. We then filtered the interaction events to only include events with different levels of intensity greater than a threshold. As shown in Table 1, the average number of methods a programmer encounters in a day is 67, or 51-83 with .95 confidence. In some cases, the programmer accessed as few as five methods, and as much as 300 methods in a day.

Filter	Avg. size of context	$\pm$ .95 conf. interval
none	67	15.6
> 0	40	5.5
> 5	17	1.0
> 10	11	0.4
> 20	6	0.1

Table 1. Average size of context.

How many times does a programmer transition to other elements?

When a programmer performs a task, many transitions to different methods are made. Evaluating transitions is important because each relocation imposes a cost in terms of cognitive load, disorientation, and time to perform search.

In Table 2, we show the number of interactions created by our programmers: some programmers developed new functionality, some were heavily navigating while debugging, while others performed incremental tasks.

A	B	C	D	E	F	G	H	I	J
16	33	100	27	93	49	35	21	11	8

Table 2. The total number of interactions (in thousands) for ten programmers A-J.

From this data, we determined how often the programmers transitioned to other classes. On average, 60% of transitions were to methods located in another class.

Finally, we analyzed how often the programmers returned to methods previously visited. On average, 95% of transitions are to methods a programmer has visited before in the same day.

Our findings suggest programmers may spend a considerable amount of time navigating source code to reacquire methods previously visited. Our programmers actively considered 51-83 methods during the day.

7.3. Context Recovery Performance

In this section we examine how well a context-recommendation system could support recovery of previously visited methods.

In our experiments, programmers needed to interact with 51-83 methods and frequently revisited these methods. If a context-recommendation system is used, *how often could a programmer have accessed a method in the context rather than via normal navigation efforts?*

The following experiment was performed: For each day, the transitions were extracted from the programmer's interaction event stream. Any event with intensity of zero was removed from the event stream. We then simulated different method replacement algorithms for differently sized contexts on the programmer transitions.

There are two precautions to consider with this evaluation technique: (1) the assumption that accessing the method from the context is easier than normal navigation, and (2) the effects of the programmers modifying their behavior when making use of the context. For (1), we took the following step to address this problem: we limited the evaluation to inter-class transitions – transitions to different classes are typically more expensive than transitioning to another method in the same class. For our data this should be sufficient because only a small percentage of interactions were direct navigations such as Goto-Definition, Goto-Reference, and Navigate-Backward; instead, programmers preferred to use tabs and the file browser. This suggests navigation from structural components is not entirely sufficient. Finally, without active maintenance of tabs in Visual Studio, the number of tabs quickly accumulates thereby increasing the time to search and locate. While (2) is a problem, qualitative analysis of the effect in similar tools is generally positive.

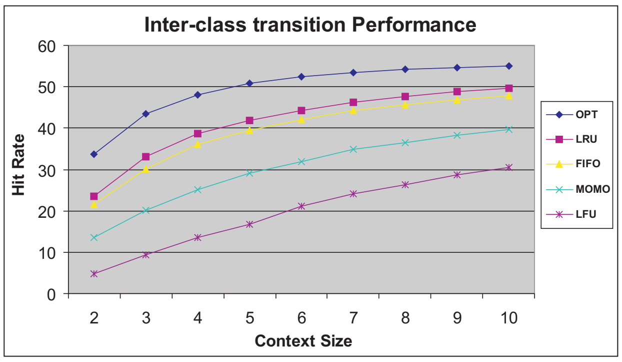

Figure 4. A comparison of different method replacement algorithms.

In Figure 4, we evaluate the different replacement algorithms in maximizing how many times the destination of an inter-class transition was present in a context. LFU performed very poorly with linear growth in respect to the context size. While MOMO did better, the performance was poor in comparison to OPT. Both FIFO and LRU performed well with LRU doing slightly better. The results of this experiment suggest temporal locality is the dominant feature in deciding the most relevant method. While momentum offers an intuitive way of examining the importance of a method, it does not perform well in periods of low intensity or searching.

In the experiment described for Table 3, the LRU method replacement algorithm was used to evaluate the recovery of previously visited methods for ten programmers A - J. Evaluating recovery is subtly different than inter-class transition hit rate. In recovery, the initial visit to a method is not penalized; instead, only the performance of previously visited methods is evaluated. From this experiment, we can see even with a small context size ($k = 4$), recovery of previously visited methods is well supported.

A	B	C	D	E	F	G	H	I	J
49.2	50.5	62.2	57.0	51.9	46.5	69.6	73.8	53.6	47.0

Table 3. Inter-class recovery transition hit rate of programmers, $k = 4$.

7.4. Context Prefetch Performance

In this section we examine the performance of recommendation systems when introduced into a context. We implemented two recommendation systems to be used with prefetch algorithms: `NEXT`, which predicts likely transitions given the current state, and `ASSOC`, which predicts associated methods given some prior methods occurring in the context.

As a baseline, we tested the accuracy of the `NEXT` system in predicting a programmer's interactions. In one experiment, we trained the systems on the first half of the day, and tested on the interactions from the second half of the day. Because recommendations cannot be actually followed, they cannot change the outcome of how the programmer interacted with the source code. This may cause some "correct" recommendations to not be counted; however, the context approach is subjected to the same conditions. What we are concerned with is evaluating the effect of extending the lifetime of recommendations and the recommendations' effects on recovery.

When combining `NEXT` and `ASSOC` with a context-recommendation, evaluating the predictive power can be difficult: with previously accessed methods largely dominating the targets of transitions, any recommendation system exploiting this information will potentially overshadow the performance of `NEXT` and `ASSOC`. To better isolate the predictive power of these systems, we used these metrics:

explore transition hit rate: Evaluates how often a prefetch predicts methods not previously visited,

prefetch recovery hit rate: Evaluates how often a prefetch predicts a previously visited method.

In the tables below, the following abbreviations are used: *class* for inter-class transition hit rate, *recovery* for inter-class recover transition hit rate, *explore* for inter-class explore transition hit rate, and *fetchrec* for inter-class prefetch recovery hit rate.

In Table 4, the performance of `NEXT` as a stand alone recommendation system is evaluated. `NEXT` is trained on the first half of the data from a day, and then tested on data from the second half of the day. The class hit rate is rather poor, resulting in only 8%-10% hits. The actual predictive power is even less when examining the explore hit rate: 5%-6%.

Context	2	3	4	5	6	7	8	9	10
recovery	9.9	10.7	11.1	11.4	11.6	11.8	11.9	12.2	12.2
explore	5.2	5.8	6.2	6.4	6.5	6.5	6.6	6.6	6.7
class	8.1	8.8	9.2	9.5	9.7	9.8	9.0	10.1	10.1

Table 4. `NEXT` recommendation system.

The next step is to evaluate how `NEXT` performs when used to generate recommendations for prefetchs of a context-recommendation system. After experimenting with different prefetch policies, we obtained the best performance by prefetching whenever the context received an inter-class transition hit. The insight into this approach is that when the programmer transitions to a new method in the context, the context should prefetch some related methods associated with that method. The number of recommendations retrieved is $\lfloor |context|/2 \rfloor$.

Context	2	3	4	5	6	7	8	9	10
recovery	22.3	35.6	39.0	42.9	42.6	44.7	44.9	44.7	44.8
explore	14.7	15.2	22.3	23.0	28.3	28.5	28.9	30.7	37.1
fetchrec	1.4	2.5	2.8	3.9	4.9	5.1	6.2	7.0	7.1
class	17.6	24.8	28.3	30.8	33.0	34.6	35.4	36.3	37.8

Table 5. `NEXT` with context-recommendation system.

In Table 5, the results from our experiment in using `NEXT` as a prefetch algorithm are presented. The overall improvement is quite good. The class hit rate for a context of size five improves from 9.5% to 30.8%. With a context of size ten, the predictions improve from 6.7% to 37.1%. (Note that in a context of size ten, `NEXT` is only giving five recommendations.)

Another experiment was performed using `ASSOC` as the prefetch algorithm. Like `NEXT`, the prefetch was performed after an inter-class transition hit and replaced half of the context. The association rules were queried using different combinations of up to four elements from the context. From the methods returned, recommendations were selected at random.

In Table 6, the results of evaluating `ASSOC` with a context-recommendation system is presented. While `NEXT` has the best performance in predicting explore transitions, `ASSOC` still results in higher overall inter-class transitions hits. This is because `ASSOC` is still experiencing better recovery rate. From this experiment, we can conclude that we can build recommendation systems that still preserve the ability to recover, while allowing the ability to introduce methods that are currently outside of the context.

Context	2	3	4	5	6	7	8	9	10
recovery	35.1	52.1	57.5	63.4	67.9	71.9	74.3	76.9	78.1
explore	5.4	7.4	10.0	10.8	10.7	10.2	10.8	10.6	10.8
fetchrec	0.6	1.6	3.2	3.6	4.7	4.7	5.4	6.6	7.3
class	18.9	28.3	33.3	37.1	40.2	42.7	44.5	45.2	46.3

Table 6. `ASSOC` with context-recommendation system.

In preliminarily analysis, it appears `NEXT` is predicting methods that will be stepping stones to the desired method. This ends up clogging the context with methods not actually being used and thus disrupting recovery.

8. Conclusion

In this paper we introduced a set of models and techniques for building a context from a history of programmer's interactions with source code. These techniques allow us to support the ability of the programmer to recover previously accessed methods.

For a context of size 2-10, we evaluated the ability to recover previously visited methods. Using different method replacement algorithms, we found we can obtain close to optimal recovery by discarding the *least recently used method*.

The summary of our evaluation of data is as follows:

- 95% of transitions were to methods a programmer has previously visited in the day.
- Using a context of size four with LRU replacement, 69% of *previously visited methods residing in another class* were able to be recovered from the context.
- The best method of recovering previously accessed methods was to discard the *least recently used method* from a context.
- The prediction power of a recommendation system may be boosted to 2-3 times the inter-class transition hit rate, and to 3-5 times the explore transition rate while retaining a superior recovery rate compared with a traditional recommendation system.

A Markov recommendation system such as `FAN` that provided five recommendations predicted 6% of explore transitions. This performance was increased to 23% when two recommendations were introduced into the context of size five on every inter-class transition hit. Using this approach, the lifetime of recommendations was increased while previously accessed methods remained easily recoverable. We conclude that inserting recommendations into a context demonstrates improved exploration of source code over traditional recommendation systems.

9. Future Work

Analyzing the interaction history of programmers has provided insight into the nature of how programmers work. We encourage other researchers to continue gathering this data to better understand how programmers comprehend and navigate source code. The following issues would be interesting research topics for future work.

Although we know the optimal performance of recovery in terms of maximizing the number of times a method is available in the context, we would like to investigate alternate measurements for algorithms which keeps methods deemed the most "important" to the programmer; possible metrics include relevance to the task or the difficulty of relocating the method.

Additionally, we would like to explore how to use contexts in other applications: in retrieving past contexts related to current tasks, and in using contexts as DOI filters in IDE interfaces and when imported by other software tools.

Acknowledgments. Spencer Rugaber, John Stasko, and our anonymous reviewers gave helpful comments on earlier revisions of this paper.

References

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[2] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Sys. J.*, 5(2):78–101, 1966.

[3] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In M. T. Elizabeth Dykstra-Erickson, editor, *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems*, pages 175–182. ACM Press, 2004.

[4] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[5] X. Fu, J. Budzik, and K. J. Hammond. Mining navigation history for recommendation. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 106–112, New York, NY, USA, 2000. ACM Press.

[6] J. Gwizdka. Timely reminders: a case study of temporal guidance in pim and email tools usage. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, pages 163–164, New York, NY, USA, 2000. ACM Press.

[7] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 3–9, New York, NY, USA, 1992. ACM Press.

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.

[9] W. Jones, H. Bruce, and S. Dumais. Keeping found things found on the web. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 119–126, New York, NY, USA, 2001. ACM Press.

[10] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[11] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa. Effective personalization based on association rule discovery from web usage data. In *WIDM '01: Proceedings of the 3rd international workshop on Web information and data management*, pages 9–15, New York, NY, USA, 2001. ACM Press.

[12] B. O'Conaill and D. Frohlich. Timespace in the workplace: dealing with interruptions. In *CHI '95: Conference companion on Human factors in computing systems*, pages 262–263, New York, NY, USA, 1995. ACM Press.

[13] J. Singer, R. Elves, and M.-A. D. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 325–334. IEEE Computer Society, 2005.

[14] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[15] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *IWPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 17, Washington, DC, USA, 1997. IEEE Computer Society.

[16] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information, 2002.

[17] I. Zayour and T. C. Lethbridge. A cognitive and user centric based approach for reverse engineering tool design. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 16. IBM Press, 2000.

[18] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04:*

Proceedings of the 26th International Conference on Software Engineering, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.