Enriching Revision History with Interactions

Chris Parnin[*] , Carsten Görg[†] , Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{vector,goerg,spencer}@cc.gatech.edu

ABSTRACT

Revision history provides a rich source of information to improve the understanding of changes made to programs, but it yields only limited insight into how these changes occurred. We explore an additional source of information – program viewing and editing history – where all historical artifacts associated with the program are included. In particular, we suggest augmenting revision histories with the interaction history of programmers. Using this additional information source enables the development of several interesting applications including an influence-recommendation system and a task-mining system. We present some results from a case study in which interaction histories from professional programmers were obtained and analyzed.

Categories and Subject Descriptors: H.4[Information Systems Applications]:Information storage and retrieval.

General Terms: Measurement.

Keywords: Interaction history, revision history, data mining.

1. INTRODUCTION

During the process of developing software, programmers leave behind traces of their intentions, tasks, and missteps. Researchers have examined how to extract these traces for the purposes of acquiring additional insight into a program's history and the developers involved in the process. Artifacts created as byproducts during the development process offer a trove of insightful information – one popular artifact is the source code revision history.

Researchers have proposed how to use revision history in applications such as program evolution or defect detection. For example, Görg and Weißgerber [2] discovered incomplete refactorings that did not preserve semantics by analyzing revision histories. Alternatively, other researchers have examined how to use revision history to gain perspective into programming activities and developers. Sliwerski *et al.* [9] examined revision histories to find changes to a previous bug fix and analyzed which day of the week contributed to the most such incidents – Friday. Mierle *et al.* [5] attempted to find correlations between a student's grade and properties extracted from the student's revision history.

Revision history can be defined as a collection of revisions to a file committed by an author through a transaction or change request. Within each transaction, several pieces of metadata can be captured: date and time, change description, and change identifier or bug fix identifier. Furthermore, information about which methods have been changed between revisions can be derived.

Revision history transcribes various snapshots of source code; however, it has limited ability in explaining how the transition between revisions occurred. Information such as what methods were frequently referenced by programmers to perform a change or the order of edits is missing. Information detailing how a revision changed can be obtained through analyzing the interaction history. *Interaction history* records a user's interactions, captured by an application such as a viewer or editor, to understand more about the user and the program. All interactions are recorded into a stream of interaction events.

In attempting to understand a program's history, all possible artifacts should be used. Interaction history provides detailed information about a programmer's activities; however, the stream-like nature of interaction history makes segmenting into meaningful sessions problematic. A reasonable approach is to use revision history as the baseline for segmenting interaction history. With this segmentation, an interaction history session can augment the associated revisions. In the other direction, revision history provides more fine-grain details about the changes made to a program, while interaction history is primarily concerned with the locations of code involved with certain interactions. In the following table, the properties of revision and interaction history are contrasted.

properties	revision history	interaction history
metadata frequency	per transaction	per event
metadata type	time, log, ID	time, target, event type

In this paper, we propose that the interaction history of a programmer interacting with an interactive development environment (IDE) can be used in conjunction with revision history to enrich current approaches. We detail how to obtain the interaction history from an IDE and explain what properties to examine. Finally, we illustrate the use of interaction history in an influence-

[*]© ACM, 2006. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will appear in the Proceedings of the 2006 International Workshop on Mining Software Repositories

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

recommendation system and a task-mining system using interaction history obtained from a case study of professional programmers.

2. INTERACTION HISTORY

2.1 Background

An *interaction history* is a record of a user's interactions with an application for the purpose of providing insight into the data as well as execution of 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 wear/read wear* [3]. *Wear* is the concept of digital objects embedding the history of its 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.

Researchers have developed recommendation systems which analyze navigation history in order to recommend that locations of interest. In `FAN` [1], navigational history is analyzed to display a list of methods that are frequently accessed next after visiting the current method. In `NavTracks` [8], navigation loops are recovered from recent navigation paths and the files related to the current method are displayed. Both `FAN` and `MYLAR` [1, 4] have used a degree of interest model based on edit and navigation frequency to indicate the 'hot spots' in source code. Finally, Schneider *et al.* [7] have used interaction history to support awareness of activities among a team.

Previous research with interaction history has focused on understanding navigation patterns and deriving simple frequency statistics. We propose a set of abstractions over interactions that allow more interesting analyses to be performed.

2.2 Abstracting Interactions

Programmers interact with source code through an IDE and revision control system. Abstractions of these interactions allow us to reason about the semantic implication of different interactions with source code entities (in this paper we assume methods).
The categories of interactions with an IDE are the following:

navigation: A command used to go to a specific location in a file such as a to method.

click: A mouse selection of a method.

edit: A change in a line of code.

inspect: An examination revealing the metadata associated with a method such as a comment or type information. This is typically accomplished by hovering the mouse over the method.

query: A search for the locations of a method.

shelve: A text editing operation on code, such as a copy or paste.

Interaction history is represented as a stream of interaction events, where each event is a tuple of method, interaction type, and timestamp. An example is as follows:

$$[(A, click, 1), (A, edit, 2), (A, copy, 3), (B, paste, 4), (C, nav, 5)]$$

When using a revision control system, programmers also interact with source code. A list of interactions would include:

revision: A set of changes made to a file.

tag: A set of metadata such as comments, bug numbers, and change packages associated with a revision.

2.3 A Visual Studio Plug-in

We built a Visual Studio plug-in called `InteractionHistoryDB` that records the interaction history of programmers using `Visual Studio`. The plug-in registers interactions exposed through the `Visual Studio` add-in interface and logged the active method targeted by the interaction.

We identified six interaction types, however, we only recorded *click*, *navigation*, *shelve*, and *edit* interactions in our experiments. Click events were recorded using a mouse-message hook. Navigation events included commands such as *goto definition*, change active tab, select a class or file, navigate from a *find-in-files* result. Edit actions were recorded by listening to events raised when a line of code was changed. The edit event is not raised until after the user changes focus from the line being edited.

2.4 Case Study

Ten employees from a defense contractor volunteered to participate in a trial use. 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 ten years old while others were in new development.

3. METADATA ANALYSIS

In analyzing program history, the questions asked about the history are generally motivated by two different perspectives:

program-oriented: Analysis focuses on retrieving code that satisfies a given query.

developer-oriented: Analysis focuses on understanding properties associated with a developer.

Detecting bad smells in code is an example of program-oriented analysis, while *determining what day of the week do the most navigations occur* is an example of developer-oriented analysis.

Regardless of perspective, the approach in analyzing the metadata of interaction history is much like revision history; however, different types of interactions are examined, and the granularity of the events are more fine. In general, revision history is a record of *what* happened, while interaction history is a record of *how* changes happen.

Localized frequency is the analysis of how events occur over a period of time. This analysis can be used in forming models of programmer interest and understanding the structure of programming activities. Later in this section, we provide examples of how interactions from interaction history can be used to enrich analysis.

3.1 Localized Frequency

The period of time or frequency that a programmer interacts with a method can be used to indicate interest. When users navigate source code, they often "thumb" through the code to locate the next method of interest. This can produce some interactions which are not desirable for analysis.

With localized frequency, a model of interest used in queries for analysis or filtering interaction data can be derived to mitigate this problem. We define this model as *intensity*.

Intensity. The intensity of an interaction with a method is *the number of prior consecutive interactions with the same method during the period of interest.*

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

Although a programmer may interact with many methods in the 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.

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. This requires a continuous evaluation of localized frequency that 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 commenced, and t_n is n steps after t_0.

Instead of having a value of zero after a transition, momentum exponentially decays the intensity. An interaction event having an intensity of 20, with a discount rate of 0.1 would have a momentum of 7.4 after ten steps and a momentum of 1.0 after 30 steps. There is a small twist: a method that is decaying can be reinvigorated when revisiting the method. In this case, the remaining momentum is accumulated with the newer intensity, and t_0 is reset to be the new time.

Momentum gives a better measure of which methods are active during a window of time; however, its continuous nature can make it more difficult to apply in some situations.

3.2 Example Queries

What term is most commonly searched? Analyzing the distribution of search terms may reveal items difficult to locate, items not immediately understood by the programmer, or items relevant to the current task. Filtering out events without query interaction types results in the following example interaction event stream:

$$[(\text{“}display\text{”}, query, 1), (\text{“}screen\text{”}, query, 56), (\text{“}display\text{”}, query, 78)]$$

In this stream, 66% of the queries were for “display” and 33% for “screen”.

What is the ratio of transitions to edits? Understanding the relationship a programmer's edits and transitions between methods in a project inspires several applications: (1) it serves as a baseline for comparing tools in experimental studies, (2) the rate of change of the navigation/edit ratio can be used to determine when a programmer is searching, and (3) it can assist in classifying tasks applied to the same project.

Consider the following event stream as an example:

$$[(A, click, 1), (A, edit, 2), (A, click, 3), (A, edit, 4), (B, click, 5)]$$

To avoid a heavily edited method from imposing too much bias on the ratio, navigations within methods (in our example (A,click,3)) are first removed, and then from the resulting stream the consecutive edits within a method (in our example (A,edit,2) and (A,edit,4)) are considered as one edit. After this preprocessing, the stream appears as follows:

$$[(A, click, 1), (A, edit, 2), (B, click, 5)]$$

The navigation/edit ratio for this stream is 2.0.

What time of day has the highest activity? Activity analysis can assist in studies of work patterns or in giving practical guidelines of when to schedule meetings.

To calculate this measure, the interaction event stream must first be segmented into different sessions corresponding to each hour of the day. Then, the length of each session belonging to the same hour is accumulated and averaged.

In data from our case study, 2-3pm was the period of highest activity for almost all programmers.

4. APPLICATIONS

In this section, we present two systems focusing on program-oriented analysis of interaction history.

4.1 An Influence-Recommendation System

Programmers frequently copy and paste code during the development process. The purpose of copying code varies: sometimes the programmer is avoiding the need to retype a variable name, or the copied code is being used as a template for a new variation.

Method A is *influenced by* method B if B contributed to the implementation of A through the importation of code. The influence set of A can be calculated from an interaction history by finding the origin of pasted code.

$$Influence(A) = \{B \mid \text{copy from B and paste to A}\}$$

In the following interaction event stream:

$$[(A, copy, 1), (B, paste, 3), (C, paste, 5), (C, copy, 6), (D, paste, 8)]$$

the influence sets are the following:

method	A	B	C	D
influence set	$\emptyset$	$\{A\}$	$\{A\}$	$\{C\}$

strength: The number of lines copied.

complexity: The cyclomatic complexity of code copied.

support: The number of occasions the method was influenced by the same method.

Finally, with the influence sets available for each method, an influence-recommendation system can be built. When a programmer is interacting with a method, the influence set can be used to recommend to the programmer items of interest in a contextual list displayed in an IDE. In preliminary analysis of our case study data, influence sets found (1) methods complementary to a process such as starting and stopping a server, (2) code duplication, and (3) methods serving as sources of examples (*e.g.* how to invoke a socket function call).

4.2 A Task-Mining System

Consider these situations:

1. During development of a major product release, the project manager anticipates the need to add support for a new system. The program developers interleave their normal programming tasks with integrating support for the new system over several months of development. Unfortunately, the project manager's gamble does not pay off; the customer decides not to use the new system. As a result, the programmers need to identify and remove the changes that were made to support the new system.

2. A program developer needs to perform an update to an established project. The source code contains millions of lines, but the update should only involve a relatively small subset. The programmer would like to explore the program as efficiently as possible in order to make the changes, but is too unfamiliar with the relevant parts of the source code to be sure how best to proceed.

The tasks facing our two users are different; however, in both cases, we have a user who is interacting with a large set of highly-structured data and attempting to solve a specific task. Further, their tasks require connecting various parts of the data (*e.g.* different files or methods) that are relevant in solving their task. Discovering or recalling these connections is costly; the connections may not be readily apparent from the structure, or only a few relevant pieces of information are needed among a large selection of data.

In the first scenario, identification of different tasks performed with the source code can ease the search for methods related to implementing the new system. To identify the distinct tasks performed by the programmers, clustering of the interaction sessions can be performed. Further, labels from revision history supplement identification of different sessions.

In the second scenario, providing recommendations for related interactions enables the programmer to focus on understanding just the relevant methods. Recommendations can be queried based on the last few interactions of the programmer with the IDE.

To mine these tasks from interaction history, the data must first be segmented into different sessions and then represented in a vector space model (VSM). The straightforward approach is to use revision history to group interaction events related to the same groups of transactions in the same session. Each session is then represented as a vector by counting the occurrences of a *method* and its *interaction type*.

Distance metrics measure the similarity of two sessions. Two common metrics are the euclidean distance and the cosine distance. The euclidean distance uses the L^2 norm or dot product of two vectors. The cosine distance measures the angle between the vectors normalized by their magnitude. In both metrics, a higher value indicates a higher similarity.

In the following example, three sessions are encoded as vectors and stored in a matrix. In the first session, “payroll” was edited five times, “print” was navigated to three times, “report” was navigated to two times. In the first session, the methods “employee”, “sales”, “dbquery” were not interacted with.

$$\begin{pmatrix} payroll.edit : 5 & employee.nav : 3 & dbquery.edit : 6 \\ print.nav : 3 & payroll.edit : 5 & employee.nav : 3 \\ report.nav : 2 & sales.edit : 1 & sales.nav : 2 \end{pmatrix}$$

After the sessions are represented in the VSM, several refinements can be made. Methods that are frequently or rarely interacted with can be weighted using measures such as tf-idf [6].

To identify distinct tasks, a standard clustering algorithm such as k-means can be performed. Reconstruction of the vector space with transformations such as independent component analysis may be necessary in order to more easily discriminate different tasks.

Using implicit query, the IDE can generate recommendations based on the closest session or centroid. If the programmer visited “employee” and edited “payroll” then an implicit query could be constructed as follows:

$$\begin{pmatrix} employee.nav : 1 \\ payroll.edit : 1 \end{pmatrix}$$

producing the following table of relevant interactions:

	v_1	v_2	v_3
dot-product	5	8	3
cosine	0.57	0.96	0.31

In this case, v_2 is returned as the most relevant interaction vector. An IDE, can now suggest editing *sales* through a degree-of-interest visualization or in a task pane.

5. CONCLUSION

In this position paper, we show that by incorporating interaction history with traditional approaches used with revision history, further insight can be gained. In program-oriented analysis, interaction history enriches analysis by introducing data previously unattainable and provides more options for making decisions in filtering data. In developer-oriented analysis, more data is available for understanding how the programmer performed tasks.

We conclude that interaction history: (1) can be easily obtained, (2) contains a rich source of interesting data, (3) offers powerful applications in recommendation systems, and (4) complements revision history as a source of insights into programmer behavior.

6. REFERENCES

[1] 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.

[2] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR’05: Proceedings of the International Workshop on Mining Software Repositories*, New York, NY, USA, 2005. ACM Press.

[3] 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.

[4] 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.

[5] K. Mierle, K. Laven, S. Roweis, and G. Wilson. Mining student cvs repositories for performance indicators. In *MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[6] G. Salton and C. Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[7] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer’s local interaction history. In *MSR ’04: Proceedings of the 2004 international workshop on Mining software repositories*, pages 106–110, 2004.

[8] 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.

[9] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories*, New York, NY, USA, 2005. ACM Press.