Understanding Interaction Differences between Newcomer and Expert Programmers

Lijie Zou and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
{lzou,migod}@uwaterloo.ca

ABSTRACT

Newcomer and expert programmers often interact with development artifacts differently. Ideally, software development tools should support these different styles of work. In this paper, we describe our investigations into the interaction difference between newcomers and experts, regarding two properties that characterize repetition of programmer interaction: temporal locality and interaction coupling recurrence. We describe our approach, research questions and planned methodology.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Productivity, programming teams*

General Terms

Developer behavior, newcomer and expert

1. INTRODUCTION

Programmers interact with software artifacts while working on maintenance tasks. Based on the characteristics of interaction, tools can be built or tuned to support programmers, such as recommending artifacts. For example, based on a Degree-Of-Interest model that favours recent interaction, Mylyn filters out artifacts that are unlikely to be immediately useful from IDE [4]. Based on small loops within recent interactions, NavTracks recommends artifacts to visit [13].

However, there has been relatively little study that focuses on how the artifact interactions of newcomer developers — what Sim et al. termed "software immigrants" [12] — differ from that of experts. By "newcomers", we mean experienced developers who are new to a given development team and/or project; the newcomer experience may be likened to that of immigrants who have arrived in a new land and must learn its language and culture [12]. Compared to experts, newcomers often lack domain expertise, and have little familiarity with the application codebase, the technical infrastructure in use, and the accepted work processes. These factors can lead to work patterns that are very different from those of experienced developers. For example, newcomers may be less efficient in looking for information, have more difficulties understanding it when they do find it, and be more error prone when making changes.

The grand goal of our work seeks to investigate these differences, and to develope smarter tools that can aid newcomers, either by anticipating their likely behavior or by giving advice. For example, if newcomers are found to have difficulty in understanding parts of the software system, then we might recommend documentation in a timely manner, before they have to ask for the help of a mentor. If a newcomer is likely to use multiple steps with overlapped knowledge in solving a problem, then we may collect information used from previous steps and recommend them in later ones. We may also help newcomers by poviding advice when interaction patterns are recognized. Patterns exist in programmer interactions and they are associated with typical behavior [22]. For newcomers who are unfamilar with these patterns, advice on how to perform them can be useful. Understanding the differences between newcomers and experts is also important for evaluating tools, including recommender systems. A given tool may perform differently only because of the difference between programmers.

In this paper, we outline some of our goals concerning the study of interaction differences between newcomers and experts. As an initial step in this research, we focus on two properties that characterize repetition in programmer interaction: *temporal locality* and *interaction coupling (IC) recurrence*. Temporal locality concerns the repeated accessing of a software artifact within a short time period (e.g., minutes) [6], whereas IC recurrence concerns repeated relation referencing over much longer periods (e.g., days) [22]. We decided to start with these paricular two properties because they have been shown to be useful in previous studies [4, 13, 6, 1, 22] and we believe that they might lead to development of smart recommender systems for software developers.

Our approach is based on mining interaction histories. Interaction histories are captured automatically using an instrumented IDE during development and analysis is performed post mortem. We plan to perform two case studies involving programmers working in realistic industrial setting.

2. RELATED WORK

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RSSE '08, November 10, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-228-3 ...$5.00.

2.1 Newcomer vs. expert

In their study of software engineering work practices, Singer et al. reported that newcomers are often less focused, and tend to spend more time studying parts that are not relevant to the task at hand [14]. In an explorative study of how newcomers adapt to new project, Sim and Holt observed several patterns that concern mentoring, program comprehension and management [12] .

As Sim and Holt noted, it is important to distinguish between newcomer and novice programmers [12]. A novice has little overall development experience, while a newcomer has no experience with a particular project; thus, newcomers may or may not also be novices.

Wiedenback found that experts are more accurate and faster than novices in performing low-level programming tasks, such as locating syntax errors and understanding code functionality [16]. Wiedenbeck and Fix found that there is link between programming experience and characteristics in their mental representation, such as mapping of code to goals and recognition of recurring patterns, or "beacons" [17]. Von Mayrhauser et al. also use the idea of beacons to explain programmer expertise in their study of program comprehension process [15].

2.2 Interaction history

Interaction history contains rich information about how program artifacts are accessed by programmers in solving maintenance tasks. Recent research suggests that it is a promising source for better understanding software development, and so for designing supporting tools. For example, it has been shown that studying interaction history can benefit team coordination in distributed projects [10]. By capturing interaction histories from industry setting, interaction coupling can be detected [1, 22], usage context can be extracted [6], and insights into software development can be obtained [21, 22, 7, 6].

A large number of tools are designed to assist programmer interaction by recommending artifacts to visit or change. The underlying rationale for recommenders is often based on relationships between artifacts that are inferred from previous use or different kinds of analyses. For example, co-changes mined from version control system can be used to recommend changes [20, 19], and structural properties from static call-graphs can be used to recommend related functions [8, 9]. Some relations are based on characteristics of interaction history itself. For example, based on temporal locality, an IDE can intelligently filter out artifacts that are unlikely to be immediately relevant [4, 5, 1]. Based on relations recovered from small loops within recent interactions, files can be recommended to visit [13].

3. RESEARCH QUESTIONS

We focus on two properties that characterize repetition in programmer interaction: temporal locality and interaction coupling recurrence.

- *Temporal locality*: The notion of temporal locality was first recognized by Denning [2] in the domain of program memory references. It refers to the reuse of specific items within relatively small time period. When temporal locality is strong, items referenced in the immediate past have a high probability of being rereferenced in the immediate future. In many other fields in computer science studying temporal locality has proved to be useful. Research has also shown that temporal locality is an important property in programmer interactions. It has been found that some expert programmers have high temporal locality in referencing source code [6]. Tools that exploit recent interaction [4, 13] also highlight the importance of this property. Previous studies of temporal locality have focussed only on viewing or referencing behavior. In this study, we also include changing behavior into the analysis, since making changes is the most important activity in software maintenance.

- *Interaction coupling (IC) recurrence*: If programmers frequently switch between two artifacts, then the two artifacts have interaction coupling (IC) [22]. Such coupling may indicate code dependency, code duplication, or something that is just "known" by a programmer. Studies have shown that analyzing interaction coupling can give insights into software development and design [22]. IC recurrence is the likelihood that an observed IC will recur again in the future. This property may be utilized to suggest relations to revisit, or recommend documents to read or experts to consult.

The reseach questions we want to ask relating to the two properties are:

- How does temporal locality of newcomers compare to that of experts?

- Does making changes to code also show strong temporal locality? If so, does it differ between newcomers and experts?

- How often do interaction couplings recur? Is IC recurrence different for newcomers and experts?

4. APPROACH

Our approach is based on mining interaction history. Programmer's interaction histories are first captured using tools plugged into the Eclipse IDE, and then various analyses are applied. Figure 1 shows the major steps:

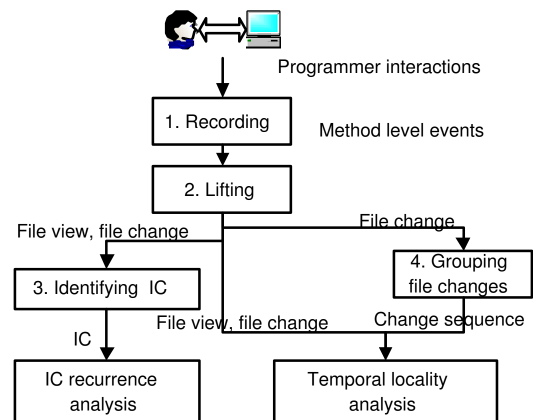

Figure 1: Approach

1. *Recording:* A plug-in records all the selecting and editing events in Eclipse while programmers are performing daily work in it. These events are at the method level, and are denoted as $(v/c, method, time)$, where v indicates that the event involved viewing an artifact, and c indicates that the artifact was changed.

2. *Lifting:* Method-level events are lifted to file level to allow for more coarsely grained analysis to be performed. Consecutive events within the same file are grouped into a sinple event. The event type at the file level is determined by whether any change event is included. If there is at least one change event, then the file level event is a file change. Otherwise, it is a file view. File level events are in the format of $(v/c, file, time)$.

3. *Identifying IC:* ICs are detected by counting contextual switches for pairs of files within some unit of analysis, e.g., a task or a day [22]. In this study, we choose the work day as the unit of analysis since it is a natural unit for developer.

4. *Grouping file changes:* When analyzing changing behavior, we focus only on file change events. Consecutive change events to the same file are grouped into one change sequence, since we consider them to be multiple steps towards the same goal. For instance, suppose we have following events after Step 2 lifting: $(v, F, t1)$, $(c, H, t2)$, $(v, G, t3)$, $(c, H, t4)$, $(c, G, t5)$. By grouping the 2nd, 4th and 5th file change events, two change sequences will be produced: $(H, t2, 2)$ and $(G, t5, 1)$, where the last number indicates how many file change events are grouped.

After these steps, we analyze temporal locality and IC recurrence.

We use the hit rate of a Least Recently Used (LRU) cache to measure temporal locality. We use this method because it is simple and was used in previous study [6]. Other measurements are also possible. For example in web traffic analysis, Zipf's distribution, entropy, stack distance and inter-reference distance have been used to characterize temporal locality [3]. We plan to study other metrics in the future.

We analyze temporal locality for both viewing and changing software artifacts. For viewing, we compute the cache hit rate for all the file view/change events (result from Step 2). For changing, we compute the hit rate for all the file change events (from Step 2) and change sequences (result from Step 4). We use two types of events for analyzing changing behavior in order to understand it from different granularities.

For interaction coupling recurrence, we check whether each coupling occurs again in following working days. Different days are used to show the recurrence in different time period.

5. METHODOLOGY

We plan to perform two case studies. Case studies are an appropriate method for our research since we want to examine programmers working on real tasks in real settings [18]. It is also suitable since our research is still at an early stage. We want to get an understanding of the area with existing research questions on mind, as well as hope to derive more hypotheses.

Our first case study will be based on the data we collected from previous studies [21]. We have already recorded interaction history of three experts and one newcomer working on their real tasks for about one month. Since it is quite expensive to collect programmer interaction in industry setting, we plan to start from this data set that is available to us. There is only one newcomer involved, so results are difficult to generalize. Our preliminary study has suggested that the newcomer has more repetitive behavior overall. The newcomer has stronger temporal locality, both in viewing and changing software artifacts. The temporal locality difference between the newcomer and experts is larger in changing artifacts than in viewing. The newcomer also has stronger IC recurrence rate; that is, interaction couplings detected from the newcomer's history are more likely to recur in the following days.

The second study will be the main case study. It will be designed to answer the research questions as presented in this paper as well as questions that may arise from the first study. We plan to recruit more newcomers and experts from industry setting, aiming for six of each. In addition to collecting data quantitivatively using tools, we will also use interviews to search for explanations of programmer behavior [11].

6. SUMMARY

We propose to study the interaction difference between newcomer and expert programmers by mining interaction history. We believe that this research can lead to a better understanding of programmer interaction and to provide inputs for improved development tool design. We are starting with two properties that characterize repetitive behavior: temporal locality and interaction coupling recurrence. While we have only just begun to formally study these properties in a realistic industrial setting, our preliminary results suggest that newcomers exhibit both stronger temporal locality and stronger interaction coupling recurrence than expert programmers.

7. REFERENCES

[1] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *VL/HCC 05: IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.

[2] P. J. Denning. The working set model for programm behavior. *Communications of the ACM*, 11:323–333, May 1968.

[3] R. Fonseca, V. Almeida, M. Crovella, and B. Abrahao. On the intrinsic locality properties of web reference streams, 2003.

[4] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168, July 2005.

[5] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th SIGSOFT international symposium on Foudations of software engineering*, pages 1–11, 2006.

[6] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC '06:*

Proceedings of the 14th Interaction Conference on Program Comprehension(ICPC 2006), 2006.

[7] C. Parnin, C. Görg, and S. Rugaber. Enriching revision history with interactions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 155–158, 2006.

[8] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, pages 11–20, Sept. 2005.

[9] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, pages 15–24, Sept. 2007.

[10] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a softare developer's local interaction history. In *MSR '04: Proceedings of the 2004 international workshop on Mining software repositories*, 2004.

[11] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25, July 1999.

[12] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 361–370, Washington, DC, USA, 1998.

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

[14] J. Singer, T. Lethbridge, N. Vinson, and A. N. An examination of software engineering work practices. In *Proceedings of CASCON'97*, pages 209–223, 1997.

[15] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the International Conference on Software Engineering*, May 1994.

[16] S. Wiedenbeck. Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, 23:383–390, 1985.

[17] S. Wiedenbeck, V. Fix, and J. Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39:793–812, 1993.

[18] R. K. Yin. *Case Study Research: Design and Methods.* Sage Publications, Thousand Oaks, CA, 2002.

[19] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[20] T. Zimmermann, P. Weisgerber, 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, 2004.

[21] L. Zou and M. W. Godfrey. An industrial case study of program artifacts viewed during maintenance tasks. In *WCRE '06: Proceedings of the 13th Working conference on reverse engineering (WCRE 2006)*, pages 71–82, Benevento, Italy, 2006.

[22] L. Zou, M. W. Godfrey, and A. E. Hassan. Detecting interaction couplings from task interaction histories. In *ICPC '07: Proceedings of the 15th Interaction Conference on Program Comprehension(ICPC 2007)*, 2007.