Does a Programmer's Activity Indicate Knowledge of Code?

Thomas Fritz[†], Gail C. Murphy[†], Emily Hill[‡]

[†]Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
{fritz,murphy}@cs.ubc.ca

[‡]Computer and Information Sciences
University of Delaware
Newark, DE, USA
hill@cis.udel.edu

ABSTRACT

The practice of software development can likely be improved if an externalized model of each programmer's knowledge of a particular code base is available. Some tools already assume a useful form of such a model can be created from data collected during development, such as expertise recommenders that use information about who has changed each file to suggest who might answer questions about particular parts of a system. In this paper, we report on an empirical study that investigates whether a programmer's activity can be used to build a model of what a programmer knows about a code base. In this study, nineteen professional Java programmers completed a series of questionnaires about the code on which they were working. These questionnaires were generated automatically and asked about program elements a programmer had worked with frequently and recently and ones that he had not. We found that a degree of interest model based on this frequency and recency of interaction can often indicate the parts of the code base for which the programmer has knowledge. We also determined a number of factors that may be used to improve the model, such as authorship of program elements, the role of elements, and the task being performed.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*

General Terms

Experimentation, Human Factors

Keywords

interaction, degree-of-interest, program structure, structural knowledge

1. INTRODUCTION

As a developer works on a system, she gains knowledge about the domain of the system, the development process used to build the system, and the design and implementation of the system [12] amongst other aspects. Since each developer's work on, and experience with, a system differs from other developers working on the same system, each developer gains a unique set of knowledge.

Various software engineering tools have attempted to form models of *who knows what* about a system automatically as a means of recommending experts for a particular aspect of a system (e.g., [9]). These expertise recommenders focus on knowledge about the system artifacts. They mine bug reports and source code revisions to determine which developers have contributed to different parts of the system. A tacit assumption with these recommenders is that interactions on a bug report or changes to the source indicates a developer's knowledge about a particular part of the system.

The idea that a developer's activity with the content of a system's artifacts is a proxy for knowledge about those artifacts seems reasonable. After all, this idea is consistent with models of how humans learn and gain knowledge. However, beyond the basic idea, what does it mean for a developer to have *knowledge* about a part of the source. Does it mean that the developer can explain what that part of the source does? Can they describe how that part of the source is structured or how it works? Answers to these questions would help direct work in expertise recommendation; for instance, by determining how much activity is necessary to gain enough knowledge to answer broad questions or by guiding the tool support needed for a supposed expert to answer a particular question. Answers to these questions could also open up the use of an activity-based model of knowledge for other purposes, such as using it for the basis of collaboration or personalizing information.

As an initial investigation into what knowledge a developer has about source code based on his or her activity, we conducted an exploratory study that focused on one kind of knowledge—knowledge about program structure—and one kind of activity—programming. In this study, a programmer's interaction with the Eclipse Integrated Development Environment (IDE)[1] was used as a description of the programmer's activity. Using a software plug-in, we monitored the interactions of nineteen industry Java programmers with their IDE over a period of five weeks. When a subject reached a threshold of interaction, our plug-in presented a questionnaire that asked about the structure of program elements with which the subject interacted. There were three thresholds, and thus three questionnaires, that a subject might be asked to answer. To ensure our study did not stress memory recall, some tool support was provided with each questionnaire to help a subject answer a question about a program element. We also interviewed thirteen of the nineteen programmers.

Through an analysis of the answers to the automatically gen-

[1] `www.eclipse.org`, verified 18/03/07

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

erated questionnaires, we found that there is a significant difference in the mean number of correct answers between program elements a subject had interacted with frequently and recently (as represented by a high degree of interest (DOI) value [6]) and program elements that were accessed less frequently and less recently. We did not find any evidence that the frequency or recency of activity alone with an element indicated knowledge. This result suggests that in a non-trivial number of cases, a programmer's activity about the program source *can* indicate one kind of knowledge, knowledge about program structure, and that a DOI model captures some of the elements for which a programmer likely has this knowledge. Through an analysis of the interview data, we determined a number of factors that influence this knowledge that are not a part of the current DOI model used, such as whether the programmer authored the element and the particular kind of task being performed by the developer when the activity took place.

We begin by situating our paper in the context of previous work (Section 2). We then discuss our exploratory empirical study (Section 3), present quantitative (Section 4) and qualitative (Section 5) results, discuss those results (Section 6) and describe limitations in our study and results (Section 7).

2. RELATED WORK

Before we undertook this study, we consulted with experts in psychology about whether studies conducted in that field would provide the evidence that activity can help indicate knowledge. Unfortunately, as we outline below, studies in psychology do not provide the foundational evidence that we would like to use to direct the building of better tools. We also looked to empirical studies of programming to find a potential link; however, we found that these studies also do not answer our initial question. Despite a lack of evidence, several tools that intend to help software developers perform their jobs more effectively assume that activity is an indicator of knowledge; we outline some of the efforts that make this assumption below to show that there is a need to investigate the question asked in this paper.

2.1 Psychology

There is a significant amount of work in psychology on knowledge. Much of this work attempts to create models that describe the different parts of knowledge, such as implicit and explicit knowledge (e.g., [4]). Ultimately, it would be desirable to understand from the neurons up how people remember and learn so that we could use those models to improve how we present information about software to programmers. Lacking this understanding, with this work, we aim to determine if we can use one kind of activity as a proxy for one kind of knowledge without trying to precisely understand how that knowledge is gained and represented in humans.

There have been a few experiments in psychology that consider programmers. Altmann [1], for instance, monitored an expert programmer performing a task for eighty minutes. He analyzed a ten minute interval using a computational simulation and studied what was likely entered into the programmer's memory on a moment-to-moment basis. His focus was to characterize near-term memory, essentially what the programmer could recall less than an hour later. We believe an understanding of what programmers can remember or know after a more significant period of time working with the same code is also important as it is more representative of the need underlying the tools that have been built (Section 2.3).

2.2 Empirical Studies of Programmers

Studies have also been conducted to explore how programmers comprehend programs, resulting in the proposal of a number of different models, such as the integrated meta-model [14]. Our focus is different, considering not the mechanism by which programmers learn about a program, but rather what they retain as knowledge as a result of the learning process. Other studies consider the type of knowledge experts have (e.g., [13]) and how experts share such knowledge with newcomers and novices (e.g., [3] and [12]). These studies do not consider how to determine *what* knowledge a programmer might have about the source; the focus of our study is to investigate one means of determining an indicator for one kind of knowledge.

2.3 Expertise Recommenders

To help make the case for the need for a better understanding of how activity relates to knowledge, we briefly survey the assumptions made in work on expertise recommenders for software developments. A few of these recommenders use profiles of expertise maintained by the experts (e.g., [7]). However, the vast majority of recommenders are based on data about the experts' activity recorded as part of artifacts created during the development. For example, Expertise Browser [9] and Expertise Recommender [7] are based on meta-data about who has made changes to which files. An approach developed to recommend who should fix a bug is based on data recorded in the bug repository system [2] as opposed to meta-data. Whether data or meta-data, the assumption is that this limited record of activity is somehow related to who knows what about a software system. These systems are able to recommend with a level of precision and recall that appears useful to developers in the limited studies conducted. We may be able to better improve the effectiveness of these kinds of tools if we develop a better understanding of what kinds of activity lead to particular kinds of knowledge.

3. STUDY

The goal of our study was to determine whether a programmer's recent interactions with the source code comprising a system correspond to his or her knowledge of the structure of the source code. We chose to focus our study on this kind of knowledge because we believe that an understanding of how parts of the system interact is needed to explain how the code works. Our hypothesis was that *the more frequently and recently a programmer has interacted with a particular source code element the higher the programmer's knowledge of that element*. (In the remainder of this paper, we use the unqualified term knowledge to mean information about a system's implementation that a developer can recall with no to minimal tool support).

3.1 Study Subjects

Our study involved industry Java programmers. We recruited these programmers from two IBM development laboratory locations. To be eligible to participate in the study, a programmer had to use Eclipse IDE in her daily work. To solicit participation, we advertised our study in a short presentation and randomly asked people at two IBM software development locations. 34 individuals showed interest in the study, split almost evenly over the two locations. From these 34, we ended up with nineteen subjects. The other fifteen individuals either did not have the appropriate tool environment, did not have sufficient time, or did not reach the predetermined thresholds of activity.

Eight of the nineteen subjects were at one development site and eleven were located at the other development site. The former group worked on two different development teams and projects. The latter group was spread evenly across six different development teams. The experience of these subjects ranged from one month to

Q1	Can you recall the name of one class or interface that is directly extended, implemented by '*TYPE*', or can you recall the name of one class that directly extends the type '*TYPE*'?
Q2	Do you know the types of the parameters that are passed to the invocation of method/constructor '*METHOD*'?
Q3	Do you know two methods that are called by method/constructor '*METHOD*'?
Q4	Can you recall one method/constructor that calls method/constructor '*METHOD*'?

Figure 1: Questions in the Questionnaire

twenty years of professional software development. The average experience was seven years (standard deviation of 6.4 years). Two of the nineteen subjects were female.

3.2 Study Method

Our overall method involved monitoring the interaction of subjects with the Eclipse IDE and prompting the subject with a questionnaire about pieces of the system structure with which he had interacted when certain thresholds of interactions were reached. Our goal was to have each subject answer approximately one questionnaire per week over a three week period.

3.2.1 Interaction Monitoring

The interactions we monitored included selections and edits of source code, and commands, such as the opening and closing of editors, views and perspectives. To ensure that the number of interactions was approximately equal between each questionnaire, we based the questionnaire prompting on the number of interaction events. We assumed an average interaction event number of 4000 per day.[2] To approximate one questionnaire per week, we thus chose 20000, 40000 and 60000 interactions as thresholds. Once the threshold was exceeded, a dialog in Eclipse opened automatically and the subjects could either work on the questionnaire or postpone it.

A questionnaire was always generated with the most recent 20000 (40000 or 60000 events); postponing a questionnaire did not affect the recency of exposure to the material asked about in the questionnaire.

3.2.2 Questionnaire Content

Each questionnaire presented to a subject contained eighteen questions about the source code elements with which the subject had interacted. Each questionnaire was generated specifically for a subject based on the interaction captured for that subject. The eighteen questions were divided into three categories of six questions. Each group of six questions contained a question about type hierarchy, two questions about type parameters, and three questions about inter- and intra-class relations (Figure 1). In each group of six questions, the second question type (Q2) was asked twice, once about a method with two parameter types and once about a method with one parameter type. Also in each group of six questions, two questions were asked about calling relationships (Q3).

[2]We made this assumption based on the average interaction events of a professional software developer who we monitored for two weeks while working on an open source tool; we reduced this programmer's interaction events by one-third to account for more meeting time in a co-located group environment.

We chose these detailed structural questions because they focus on interactions between elements and because we could determine the correctness of answers to these questions automatically. We discuss the rationale for this choice in Section 6.

A questionnaire asked these six questions for three different sets of elements: one set with high DOI, one set with medium DOI and one set with low DOI values (Section 3.2.3). To avoid learning effects, the method and type elements were chosen so that no element was asked about twice (over one questionnaire and over all questionnaires).

3.2.3 Degree of Interest

We use a real number value to represent the amount of activity a programmer has had recently with a particular program element. This real number value represents the degree of interest (DOI) of the element [6]; it is a combination of two components, a frequency of how many interactions a programmer has had with the element and a recency that decays the DOI value based on the number of interactions a programmer has had with the source since the last interaction with the element of interest. The DOI of an element starts at a positive value with the first interaction. If a programmer continues to interact with that element, its DOI will rise. If a programmer ceases to interact with the element, its DOI will gradually decay until a programmer again begins to interact with it. Different kinds of events contribute different scaled values to the DOI of an element; for instance, selections of an element contribute less to DOI than edits of an element. The DOI function used in this study has been used successfully as part of the Eclipse Mylyn[3] project for approximately eighteen months, which has approximately tens of thousands of users.

3.3 Study Support

We implemented the support for the study as an Eclipse feature[4] composed of three main components: a monitor built on the Mylyn monitor [10], a Mylyn component for computing DOI values [6], and the questionnaire component. We focus our description here on the questionnaire component.

When a threshold is exceeded, the questionnaire component must determine the elements with high, medium and low DOI values and generate the questionnaire. This component generates two lists based on the target in each interaction event being considered: one for all methods and one for all type elements. The component takes into account elements that were touched (i.e., selected or edited) at least three times to avoid elements touched by accident. Furthermore, elements that were used in former questionnaires are excluded from the list.

Each list is then sorted according to the DOI values. The 20% of elements with the highest degrees of interest, the 20% in the middle and the 20% of elements with the lowest degrees of interest in the sorted list are considered as the groups of elements with high, medium and low degrees of interest.

To form the questions, elements were randomly drawn from the groups of high, medium and low DOI. We placed constraints on the elements used for a question to make sure answers were possible. For Q1, the target element had to have at least one type it extends/implements or it is extended/implemented by. For Q2, one target method element had to have at least two parameter types and one had to have at least one parameter type. For Q3, each of the two target method elements had to have at least two method calls in their method body. For Q4, the target method element had to have

[3]`www.eclipse.org/mylyn`, verified 13/06/07

[4]The subjects used a variety of versions of Eclipse from 3.2.1 to 3.3M4.

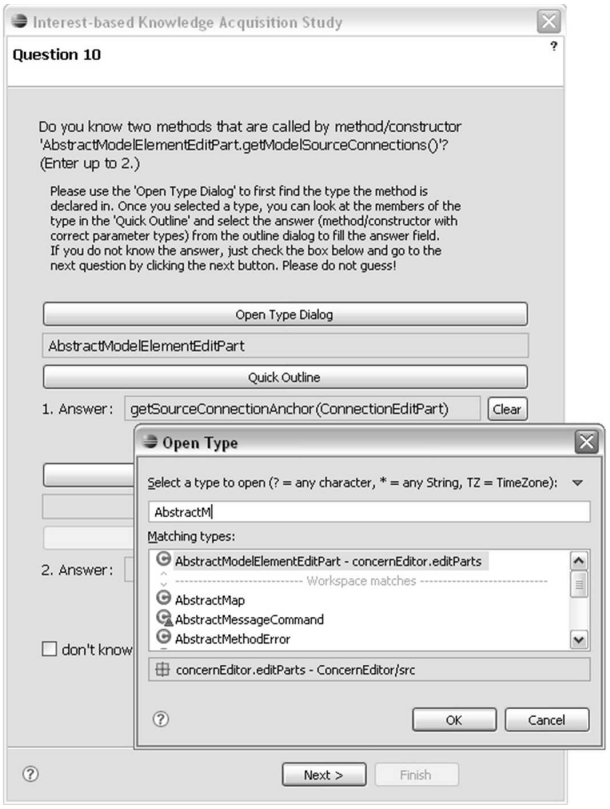


Figure 2: Questionnaire with Open Type Action

at least one method it was called by directly. To determine if an element met the requirements, the questionnaire component used the Eclipse Java Development Tools (JDT). Specifically, the component used the search engine of JDT to discover called-by and extended-by relationships and the abstract syntax tree parser to retrieve information about calling relationships, parameter types and extended and implemented types.

Once eighteen suitable elements were determined, the questionnaire component opened a wizard dialog that had one page for each question. To avoid mistakes in recalling the exact name of a type or method signature and to provide the subject with familiar features of Eclipse, our tool provided the open type action and the outline view of Eclipse. The open type action opens up a dialog and gives the programmer the chance to look for a type via a regular expression (Figure 2). The outline view tool lists all elements declared in a class; it was provided for questions in which a method name was to be entered as an answer. The answer fields of the question pages could be filled by selecting an element in either the open type view or the outline view. This approach helped eliminate misspelling and helped facilitate the answering process.

3.4 Result Scoring

Twice per week, we visited each subject and asked whether or not she or he had completed a questionnaire. If so, we collected the relevant data that contained the questions, the subject's answers to the questions and for each question all possible answers found with JDT; the possible answers were stored by the questionnaire component on the subject's computer at the time the questionnaire was administered. We then analyzed the results manually by comparing the subject's answer with all possible recorded answers for that question to detect if the given answer was correct. To ensure our tool also found all possible answers, we compared our results to the actual code bases on a subject's computer for a random sample of subjects.[5]

We then summed up the number of correct answers for each group of six questions (low, medium, high). As mentioned above, in each group of six questions, Q1 and Q4 were asked once with one possible answer each. Q2 was asked twice, once about a method with two parameter types and once with one parameter type, resulting in two possible answers in the first case and one possible answer in the second case. Q3 was asked twice with two possible answers each time. Therefore, the correct answer score for each group was between 0 and 9. For each questionnaire, we thus computed three values ranging from 0 to 9, one for the group of questions about elements with low DOI, one for medium and one for high DOI.

3.5 Interaction Levels

There was a substantial difference in the period of time it took each programmer to reach the first threshold. Several developers reached the first threshold after three days, the second after six and the third after around nine days. Other subjects took twelve days of programming to reach the first threshold. The mean time for reaching the first threshold was 7 with a standard deviation of 3.10.

3.6 Operational Problems

We also faced several operational problems when doing our study. Eight subjects that had completed the first questionnaire did not reach the second questionnaire and only eight subjects reached the final questionnaire. This attrition seems to be for one of three reasons. First, some subjects accidentally uninstalled our plug-in by installing a new version of Eclipse and deleting the old workspace in which the data was stored. Second, there was insufficient time from when a subject joined the study to the end of the study period to collect the results. Third, a questionnaire could not be created. For example, two subjects reached the second interaction threshold without having completed the first questionnaire as the study plug-in was unable to create the first questionnaire due to interaction with an insufficient number of elements to fulfill the criteria for creating the first questionnaire.

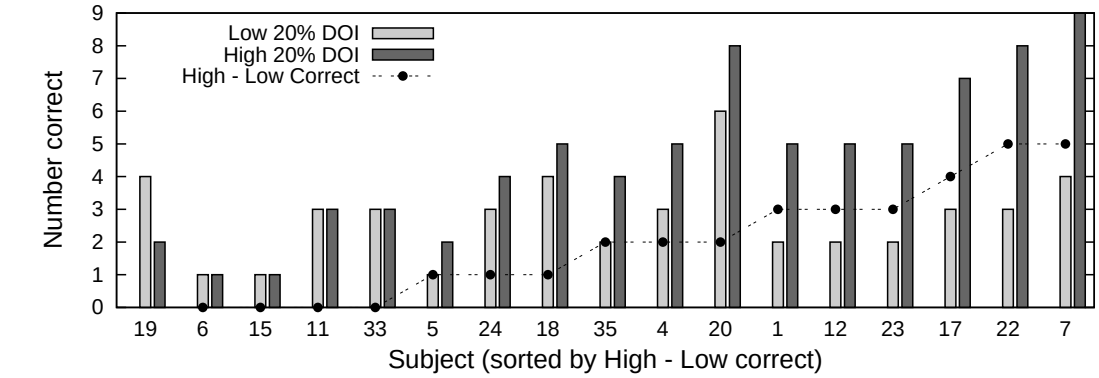
4. QUANTITATIVE RESULTS

To evaluate our main hypothesis of whether the frequency and recency of a programmer's interaction with particular parts of the source indicates knowledge of that source, we conducted within subject statistical tests. Per subject, we paired the number of correct answers for the six elements with a low DOI within a questionnaire with the number of correct answers for the six elements with a high DOI within the same questionnaire. We did not consider the elements with medium DOI in the questionnaires because the range of DOI values for those elements was very close to the range for high and low DOI elements. Please note that in presenting the data we have altered genders so as not to identify the subjects.

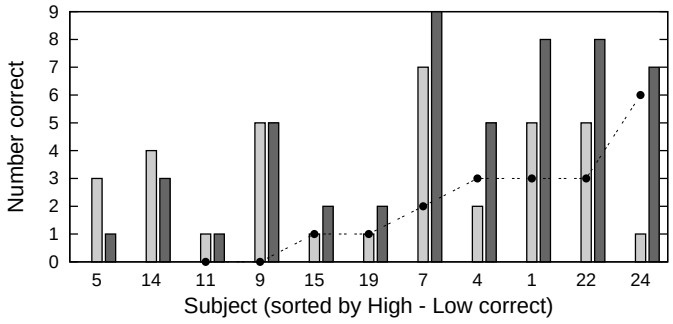
4.1 DOI and Knowledge

We depict the results of correct answers for the elements of low and high DOI for each subject for each of the questionnaires in Figure 3. We have arranged the results per subject so that the difference between the correct answers for high DOI and low DOI elements increases from left to right; the differences are shown in the trend line overlying the histogram. When the line is visible above zero,

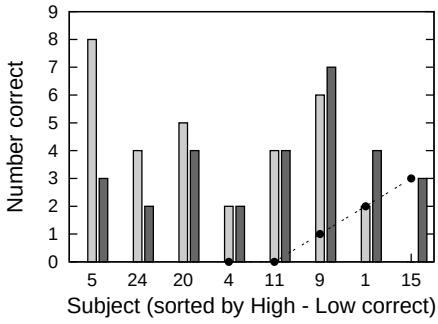
[5]We performed this extra check to account for any differences on the subjects' machines from our test environment.



(a) First Questionnaire



(b) Second Questionnaire



(c) Third Questionnaire

Figure 3: Correct Answers by Subject for Questionnaire 1, 2 and 3

it is possible to see how many subjects had more high DOI correct answers than low DOI correct answers. For the subjects where the trend line is above zero, the data supports our hypothesis that a programmer's activity, modeled by DOI, indicates knowledge.

To determine whether there was statistical significance in the mean difference between correct answers for high and low DOI elements, we performed paired t-tests by pairing the number of correct answers for high and low DOI elements by subject. The difference in the number of correct answers for each questionnaire passed the Kolmogorov-Smirnov and the Shapiro-Wilko normality tests.

4.1.1 First Questionnaire

Seventeen subjects answered the first questionnaire.[6] Using a paired t-test, we found that the mean high DOI correct is significantly[7] higher than the mean low DOI correct per subject (two-tailed $p = 0.0016$, $DF = 16$[8], mean difference $= 1.7647$).

In the results of this questionnaire it is interesting to see that all subjects [S19, S6, S15, S11, S33] who had more or the same number of correct answers for elements with low DOI as for elements with high DOI said in subsequent interviews that they were working on changes crosscutting code written by other programmers. Furthermore, these were the only subjects that described their work in this way for the first questionnaire. Subject S19 also said that just shortly before the study started he had to understand the behavior of all six elements with low DOI, causing him to answer the questions for the low DOI elements correctly. In addition, when considering only subjects with a positive difference, the subjects with less than or equal to one and a half years of professional experience [S5, S24, S18, S35] have a smaller difference in the number of correct answers than the ones with three and a half years or more.

4.1.2 Second Questionnaire

Eleven subjects reached the threshold for the second questionnaire. For the data provided by these subjects, the two-tailed P value for the paired t-test is 0.0578 and is thus not significant at the 95% significance level ($DF = 10$). We hypothesize that this lack of significance is from the lack of data: there are only eleven observations in the second questionnaire versus seventeen from the first.

For this questionnaire, five subjects [S14, S11, S9, S15, S19] stated that they were working on changes crosscutting other programmers' code and similar to the first questionnaire, their difference in correct answers between low and high is very close to zero. With the exception of subject S5, these five subjects are also the leftmost on the graph ordered by the difference and there is no other subject with a smaller difference. When looking only at the subjects with a positive difference, all subjects that have less than one year of professional working experience [S15, S19], except S24, have a smaller difference than the subjects with more than seven years of experience [S7, S4, S1, S22].

[6] Although the interaction of two more subjects caused the subjects to pass the first threshold, the study plug-in did not find enough elements in their interactions to create the questionnaire.

[7] We consider results to be statistically significant with $p < 0.05$.

[8] The degrees of freedom (DF) represents the number of observations used to estimate a parameter (for example, the differences between mean low and high DOI correct per subject). In this case, $DF =$ number of subjects -1.

4.1.3 Third Questionnaire

Only eight subjects reached the third threshold. Performing the paired t-test on the results of this questionnaire did not deliver a significant result.

For this questionnaire, we saw more evidence of individual factors influencing the results. Subject S24 stated that the third questionnaire was the hardest because there were hardly any questions about elements that were written by her. In fact, most elements with a high DOI for that questionnaire seemed to gain a high DOI because she stepped through them a lot when debugging. Subject S4 and S9 stated that all six high DOI elements asked about in the questionnaire were not written by them and they thought it was interesting and even "surprising" [S9] that those elements had a high DOI. Similarly to S24, subjects S4 and S9 stated that they had interacted with several of the high DOI elements as a step of the debugging process. Furthermore, these two subjects said they should have known better the six elements with low DOI because they wrote or edited those elements significantly.

4.1.4 Summary of Questionnaire Results

Assuming the answers from a given subject are independent across questionnaires, we tested whether the mean high DOI correct for a subject is significantly different from the mean low DOI correct. Using a paired t-test, we found that the mean high DOI correct is significantly higher than the mean low DOI correct per subject ($p = 0.0024$, $DF = 35$, mean difference $= 1.22$).

Although on average a subject only has about one more correct answer for high DOI questions than low DOI questions, this result supports our hypothesis that a programmer is more likely to have knowledge of program elements with which he or she has frequently and recently interacted.

4.2 Impact of DOI Components

The DOI value is an aggregation of multiple components. To investigate the impact of several primary DOI components, we plotted all of the elements to which one of the subjects had a correct answer with respect to its recency (one component) and the number of selects and edits (a second component) in Figure 4(a). Figure 4(b) shows a similar plot for elements to which a subject had an incorrect answer. The axes in each of these plots use a logarithmic scale.

These figures show that there appears to be no trend in correct and incorrect answers for the components of DOI aggregated over all subjects. If there was a trend, we would expect some cluster to appear in Figure 4(a) that does not occur in Figure 4(b). We also looked for a trend between selects and correctness or between edits and correctness. However, plotting the correct elements with respect to selects (x-axis) and edits (y-axis) and comparing it to the incorrect elements gives us the same result: there is no trend. Since by looking at the overall correct and incorrect answers the data might not take into account individual differences of the subjects we also broke the data down by subject. However, these plots also showed no visible trend. For example, Figure 5 shows a box plot that has the information about the recency of correct and incorrect answers per subject. There seems to be a slight trend towards correct answers being more recent, but it is not significant. Regression analysis of correct answers, edits, selects and recency yielded no further insights into the data.

4.3 Further Results

We also considered a number of other factors that might impact the results.

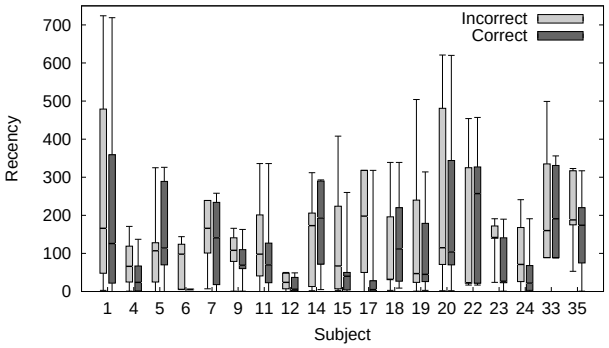


Figure 5: Recency for correct/incorrect per subject

Effect of Crosscutting Code Investigation.

As mentioned above for the first and second questionnaires, the subjects who stated that they were working on tasks involving code which crosscut the code of other developers had a smaller difference between correct answers for high and low DOI elements. Considering the data from all three questionnaires and taking the mean of the total number of correct answers per subject over low, medium and high DOI, developers who were working on crosscutting code also had significantly less correct answers (pooled two-sample t-test: $p = 0.0252$, mean difference $= 4.0301$).

Correctness Tendencies.

Subjects that had more correct answers for elements with a high DOI also tended to have more correct answers for elements with a low DOI. There was a significant correlation for both questionnaire one (Pearons's $r = 0.61623$, $p = 0.0084$) and questionnaire two ($r = 0.66469$, $p = 0.0257$). As there are few data points for the third questionnaire, we did not include it in this analysis.

Experience and Knowledge.

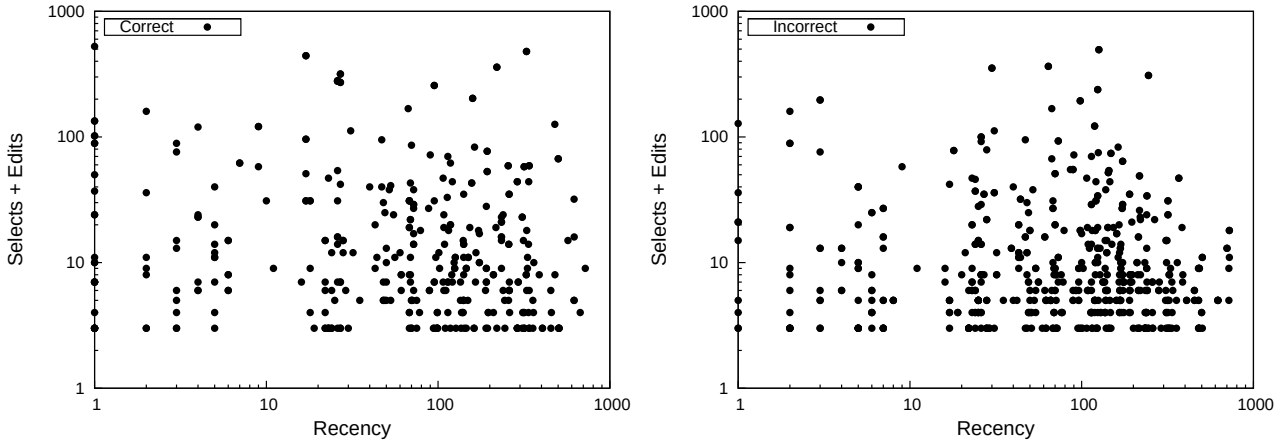
Looking at Figures 3(a) and 3(b), it seems to be the case that when looking only at the subjects with a positive difference, experience has an influence on the results (Section 4.1.1 and 4.1.2). However, we did not find a significant difference statistically that supported this hypothesis, which may be due to a lack of data after filtering those subjects whose results show a difference less than or equal to zero.

Different Knowledge for Different Question Types.

To see if there are differences in the knowledge about certain questions, we looked at all incorrect and correct answers separately, aggregating them by the type of question (i.e., type hierarchy (Q1), parameter type (Q2), called-by- (Q3), or calling-relationships (Q4). We found that there is no significant difference in the results based on the particular question that subjects were asked. Even though several subjects suggested that some question types were particularly difficult during the interviews, we can not find any quantitative support for these observations.

DOI Consistency.

Over all subjects, the mean of the DOI values for the elements in each group (low, medium, high) were in a close range. This result holds for the data from all three interaction intervals. Thus, even though subjects differ in the systems on which they are working and their tasks, the range of DOI values is similar. No matter which subject we consider, if we take the 20% of elements with



(a) Correct (b) Incorrect

Figure 4: Selects+Edits versus Recency

highest, medium or lowest DOI for a certain interaction interval (e.g., 20000 events), the DOI values for the set of elements are in a similar range.

Programming Intensity.

The number of interactions per day per subject ranged from 1850 to 8550. A reasonable assumption might be that the more a subject programs in a day, the better the subject would know the code. However, we do not see any significant impact on the results despite this large difference.

5. QUALITATIVE RESULTS

We were able to interview in-person thirteen of the nineteen subjects after each had completed the questionnaire portion of the study.[9] Each interview took between five and thirty minutes depending upon the time available from the subject. We also received some data in response to email from subjects S1, S6 and S7.

Each interview started with some general questions (see Table 1), followed by a variety of questions based on a subject's response to previous questions.

In this section, we present this qualitative data, describing where it corroborates, how it explains, and where it differs from the quantitative data.

Table 1: General Interview Questions

1	Do you think that the correctness of your answers reflects what you know about those elements?
2	Do you think there are some elements for which you should have known the correct answer and/or are there some elements where you are surprised that you knew the correct answer?
3	Why do you think you knew more about the elements that were answered correctly than about the other elements?
4	What kind of knowledge do you think you keep most in your mind over a long and/or short period of time?

[9]Interviews were conducted with S4, S5, S9, S11, S14, S15, S17, S18, S19, S20, S22, S24, and S33.

5.1 Subjects' View of the Results

We wanted to know whether the results of our study reflect what a subject believes he or she knows about the elements asked about in the questionnaires. To investigate this issue, we showed each subject a list of the program elements that appeared in all of the questionnaires that the subject had completed; this list indicated for which of the elements the subject answered the question(s) correctly. We then asked the subject if the split between elements for which he or she answered questions correctly and those answered incorrectly reflects what he or she knows about the elements. We then went over each element, asking the subject whether the correctness of the answer surprised him or her.

Overall, eleven of the sixteen subjects (69%) whom we interviewed stated that the correct versus incorrect answers represented "fairly well" what each subject knew about the code and in one case, was "exactly" what the subject knew [S20]. Two subjects (12%) stated that the first and second questionnaire was fairly reflective but the third questionnaire was not. Three subjects [S5, S19, S33] (19%) stated that she or he should have known the correct answer to some elements because she or he wrote the code. These three subjects also expressed surprise about having a correct answer to several elements.

5.2 Subjects' View on Their Knowledge

In the interviews, we also asked what kind of knowledge a subject thought she had about the program elements and what the subject thought about the questions. Most subjects [S1, S4, S7, S14, S18, S19, S22, S24, S33] (56%) responded that he or she would know what a particular method asked about does, but that the questions in the questionnaires were too detailed. Three subjects [S14, S18, S33] stated he or she would know the flow of control in the program, but he or she would not necessarily know the direct calls, only that there was some collaboration between two elements. As one subject stated, "remembering finer details isn't my strong point" [S19].

5.3 Influence of Authorship and Editing

All subjects stated that he or she knows more about elements he or she authored. When we asked the subjects about the questions they answered correctly, they mostly stated those elements were the ones they had authored. As one subject [20] explained,

> when you write your own code you follow your own patterns so it is easier to know afterwards, [...], you can tell how you would have done it.

Two of the subjects each further stated that most of her correct answers occurred for code she wrote recently [S1, S14]. One subject [S15] said that authoring the (Java) classes would probably cause one to know those classes better for one to two months. Another subject [S1] stated the opposite, saying,

> I would say that global knowledge of the system is maintained over a longer period of time but the specifics of each method implementation deteriorates quite quickly [...] if I was asked the same questions now [2.5 weeks afterwards], I would get most of them wrong.

Overall, there was a large discrepancy about the period of time that subjects thought they would know about code that he or she authored from three months [S22] to one and a half to two years ago [S5, S11]. These latter two subjects each noted that his knowledge was dependent on how long it took him to write the code—the longer the authoring time the better the knowledge—and whether he was actively maintaining the code. Several subjects each noted that he or she has to "work with code [continuously] otherwise I forget after a while [1 month]" [S18] or as subject S7 stated,

> Areas that require adjustments as features develop and new scenarios get fleshed out are the areas I know best. Areas that hardly change once initially written are areas I have trouble recalling.

Often, if the code was not authored by the subject, the subject would know the code only if it was visited shortly before the questionnaire [S20, S22]. One subject thought that if the code was visited longer ago than 24 hours before the questionnaire, the chance of getting the correct answer decreased [S11].

5.4 Influence of Code Stability

The stability of the code also has an influence on how long a subject knows about the program elements. Both subjects S11 and S5 were authoring code at a low level of the system; this code needed to be robust and does not change often. These two subjects each felt they had good knowledge of the code one and a half years after the code's creation. Other subjects, who were working on code that was changed more frequently, stated that they would not know their code more than a couple of months.

5.5 Influence of Program Elements

Several subjects [S1, S7, S9, S11, S15, S19, S22] (43%) each mentioned that he knew more about elements that played a more important role in the code. For instance, if a class was a hub of an API, the subjects [S1, S11, S22] would know the correct answers to questions posed, whereas internal code was less known. Another subject [S15] mentioned that you have to understand crucial methods to know what is going on, so you spend more time on them. Subjects also stated that they would know the abstract classes on the top of the type hierarchy or the root super class in general but they did not know about intermediate classes [S9, S19]. Furthermore, if a class was part of a test, it seemed not to be as important and therefore not to be known as well as other code [S7].

An interesting factor was also the size and simplicity of classes. Several subjects agreed that it is easier to know and remember smaller classes because the "structure is easier to understand" [S4, S14, S22, S24]. However, another subject stated that he did not know the answer to a question because the class was very small so he forgot quickly [S20].

5.6 Influence of Tasks

When we interviewed subjects about the elements, they would often refer to a set of elements as a task or talk about a general task. One subject even remembered the overall tasks three weeks after the actual questionnaire [S11].

Some subjects identified the kind of tasks undertaken over the period of the questionnaire as a reason for not having a lot of correct answers. From one [S11], "I was into code all over this place trying to thread through some of the stuff I was working on". Another [S33] explained that he fixed a lot of small bugs and explored a lot of code very briefly that he has not written and therefore he did not know the answers. Yet another [S9] explained that he worked on crosscutting changes in other people's code. When working on other people's code, subjects described that they were focusing more on getting it to work than understanding how it works [S14, S15, S19]. One [S1] also noted that if the task was to create a new feature, she knew the code more than if it was a refactoring task.

5.7 Influence of Short-term Activity

Similarly to the overall task, the activity undertaken to complete a task in the short-term with just a small subset of elements influenced knowledge of the elements. For instance, one subject [S11] said,

> It depends upon what you are doing, if you are fixing a bug you are concentrating directly in there but if you are just adding an extra parameter to pass it through to something deeper you don't know a whole lot about what's going on in there.

For nine subjects [S4, S9, S14, S15, S19, S20, S22, S24, S33] (56%), debugging activity heavily influenced the subjects responses to the questionnaires because elements appeared in the questionnaire as a result of activity stepping through the elements repeatedly during debugging. This activity was intense but did not consider the structure or functionality of the element. For instance, one subject [S24] described that the debugging influenced her third questionnaire substantially as most of the elements asked about in that questionnaire were the result of debug steps but were not representative of elements she was working with. A similar situation occurred when going through lists of search results.

5.8 Influence of the IDE

Integrated development environments, such as Eclipse, provide substantial support to find structural information about code. Four subjects [S6, S14, S15, S22] each stated that he relies heavily on these structural determination tools, "I live by the call hierarchy view" [S22]. These subjects also noted that the presence of these tools likely causes them to remember less about the kinds of information that can be retrieved fairly easily and quickly [S15],

> in a VI text editor I would probably know more about the actual calls but in Eclipse I have the JDT support to help me.

Although our questionnaires provided access to some tools, for finding types and methods, the lack of access to all tools created a "disconnect feeling" [S14] to the code.

5.9 Influence of Code Patterns

Patterns in code facilitate knowledge for some subjects [S11, S22]. As one [S22] stated, "I know what this method does because it always does the same for each class it is in". When showing one correct element to one subject [S11], she was surprised to have known the answer, but once she thought about it she said, "I did

not know this well [but] that's our pattern for writing some of the tests". This situation also arose for another subject [S7] who stated that "a lot of our method signatures contain common data structures which are easy to recall [patterns]" [S7]. These common patterns are similar to clichés [11].

6. DISCUSSION

We begin a discussion of our study with a summary of results. We then discuss what kind of knowledge matters and how we gauge knowledge. Finally, we present how these results might be applied to improve the effectiveness of programmers through improved tool support.

6.1 Summary of Results

The quantitative data provides evidence that the activity described by the degree of interest (DOI) function is a useful indicator of a programmer's knowledge about the structure of code. Our analysis also shows that splitting the DOI into the components of frequency of access and recency of access does not provide results that are statistically significant. Although our results show that DOI is a good model for indicating structural knowledge, our study cannot tell us at what threshold value DOI indicates this kind of knowledge.

The qualitative information we gained through interviews suggests that additional factors should be used to augment DOI to gain a better indicator of program structure knowledge from activity:

Authorship A programmer knows more about program elements he or she authored.

Authorship Duration The more time and effort a programmer spends creating an element, the better the programmer's knowledge about the element.

Code Stability The fewer changes that are made to the code, the longer the programmer has knowledge about it.

Code Patterns The more code patterns that are used and reused, the easier it is for a programmer to infer knowledge about those elements.

Role The more important the role of an element in the program, the better the knowledge about it.

Task Locality The more a programmer must interact with other programmers' code, the less DOI corresponds to knowledge.

Work Experience The higher the professional work experience of the programmer, the better DOI corresponds with knowledge.

Activity Short-term kinds of activity influence the DOI values of elements and can skew the correlation between activity and knowledge; for instance, when debugging.

Some of this information is easier to factor into a model of activity than other information. For example, authorship information can be easily gained through a link to the source revision repository. Similarly, task locality might be determined through source repository information. On the other hand, the role of an element may be more difficult to determine reliably.

6.2 What Knowledge Matters?

For our study, we only looked at one particular kind of knowledge — detailed structural knowledge of the source. We argue that focusing on detailed structural knowledge is a reasonable starting point for investigating knowledge as it is possible to check the correctness of answers objectively. We provided the participants access to only limited tooling to help answer the questions. We argue this constraint is a reasonable starting point to try to distinguish knowledge from tool expertise. We gained access to some information about other kinds of knowledge through the interviews conducted with the subjects. Our study is clearly preliminary; much more investigation is needed to gain an understanding of how and what programmers know about the source code and systems with which they work.

6.3 Gauging Knowledge

Our questionnaire contained questions about four different types of information: type hierarchy, type parameters, inter- and intra-class relations. We settled on these question types after a pretesting phase in which we asked several software developers which questions they thought would represent code-level knowledge. Knowledge might be better gauged by including a question type about the "flow of control" as stated by several subjects. However, we chose not to use such a question type as it is not possible to verify answers to such a question automatically.

6.4 Using Activity as a Knowledge Indicator

The results of our study help provide a foundation for current approaches to expertise recommenders, which suggest who has expertise in particular parts of the program. Most of these approaches (e.g., Expertise Recommender [7], EEL [8] and Expertise Browser [9]) make recommendations based on commits to source code repositories. As programmers in our study commented that authorship of code is a significant factor in their knowledge of the code, existing expertise recommenders build on a solid foundation.

The results of our study also suggest ways to augment existing expertise recommenders. For instance, activity traces might be used to account for other means of gaining knowledge of code other than authorship. The results also suggest that recent change and initial authorship should be weighed higher than changes made between those two points when ranking expertise.

As another example, models of individual programmer's knowledge of a source code base could be aggregated to the team level to suggest pro-actively who should be coordinating with each other. Cateldo and Herbsleb, for instance, showed that congruence between who should be coordinating and who actually did reduced the resolution time of modification requests to a software system [5].

7. THREATS TO VALIDITY

The validity of our results is threatened by several factors. A primary threat to the validity of our study is the number of subjects. In particular, for the second and third questionnaire, the number of subjects that completed the questionnaire is small, possibly skewing the results as we begin to look at a much smaller set of programmers.

The validity of our results is also threatened by undertaking the study in situ rather than in a laboratory. In situ, there are many variables for which we cannot account, such as the type of work being performed by the subject. This lack of control shows up in several ways in our results. For example, for subject S24, the number of correct answers for low and high changed substantially over the three questionnaires (Figure 3) because of the nature of her activity

during the different periods. As another example, subject S5, who had substantially more correct answers for the low DOI elements in the second and third questionnaire than for the high DOI elements, stated that for these questionnaires, there were more elements that were written by him in the low DOI elements than in the high DOI elements. Since he stated he believed he knew everything he wrote, this provides a possible explanation for his results for these questionnaires.

The validity of our results are also threatened by measuring activity with the source only through the programmer's interaction with the development environment. We chose to monitor the IDE because it is a common tool used by all programmers. Other forms of activity, such as design meetings, are much more difficult to monitor and to conceive of using as a basis for subsequent tools. Although our monitoring of the IDE was extensive, it was aimed at certain mechanisms that provide good coverage of textual editing, selections, and so on. Our monitoring misses interactions through graphical editors which some of the subjects may have been using. In these cases, the DOI values we assigned would not be representative of the actual interaction and activity of the subject with the source. The interviews we conducted with the subjects did not highlight any of these issues as seriously compromising the study.

8. CONCLUSION

All too often in software engineering, processes and tools are proposed to enhance the effectiveness of software developers that make tacit assumptions about the nature of software development work. For instance, there is a tacit assumption in the work on expertise recommenders that authorship is a strong determinant in who knows what about a system's source code. In this paper, we present the results of a study about whether the interaction of a programmer with source code indicates which parts of the source code the programmer knows. In this study, a programmer's knowledge about source is gauged by his or her ability to answer questions about the structure of the source with limited tool support. In our study of nineteen professional Java programmers, we found that the frequency and recency of interaction of a programmer with parts of the source does indicate which parts of the source a programmer knows. We also found through interviews of the programmers a number of other factors to consider including in a model of knowledge based on activity, such as authorship, the role of the source in the system, and the programmer's task. This information can be used to argue the assumptions underlying some tools intended to improve software development, such as expertise recommenders, and can be used to enhance the tools developed in the future, such as IDE views tailored to an individual programmer based on their experience with the source.

9. ACKNOWLEDGMENTS

This work was supported by IBM, the IBM Ottawa Center for Advanced Studies and NSERC. We thank all the study subjects for their participation and Terry Hon and Jelena Sirovljević for their reviews.

10. REFERENCES

[1] E. M. Altmann. Near-term memory in programming: a simulation-based analysis. *International Journal of Human Computer Studies*, 54(2):189–210, 2001.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, 2006.

[3] L. M. Berlin. Beyond program understanding: A look at programming expertise in industry. In C. R. C. Jean C. Scholtz and J. C. Spohrer, editors, *Proc. of the Fifth Workshop on Empirical Studies of Programmers*, pages 6–25, 1993.

[4] N. R. Carlson, W. Buskist, M. E. Enzle, and C. D. Heth. *Psychology: the Science of Behaviour*. 2005.

[5] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 353–362, 2006.

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

[7] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240, 2000.

[8] S. Minto and G. C. Murphy. Recommending emergent teams. In *Proc. of the International Workshop on Mining Software Repositories (MSR)*, 2007.

[9] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[10] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[11] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[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, 1998.

[13] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984.

[14] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of 16th International Conference on Software Engineering*, pages 39–48, 1994.