A Degree-of-Knowledge Model to Capture Source Code Familiarity

Thomas Fritz, Jingwen Ou, Gail C. Murphy and Emerson Murphy-Hill

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

ABSTRACT

The size and high rate of change of source code comprising a software system make it difficult for software developers to keep up with who on the team knows about particular parts of the code. Existing approaches to this problem are based solely on authorship of code. In this paper, we present data from two professional software development teams to show that both authorship and interaction information about how a developer interacts with the code are important in characterizing a developer's knowledge of code. We introduce the degree-of-knowledge model that computes automatically a real value for each source code element based on both authorship and interaction information. We show that the degree-of-knowledge model can provide better results than an existing expertise finding approach and also report on case studies of the use of the model to support knowledge transfer and to identify changes of interest.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Human Factors

Keywords

expertise, authorship, degree-of-interest, interaction, degree-of-knowledge, onboarding, recommendation

1. INTRODUCTION

Software developers working with source code face a deluge of information daily. The development environments they use provide fast access to the many (often millions of) lines of code comprising the systems on which they work. The hard work of their teammates often results in a high rate of change in that code. For a professional software development team we studied, each developer was, on average, accepting changes to over one thousand source code elements per day from other team members into their environment (Section 3).

The large flux in the source can make it difficult to know which team member is familiar with which part of the code. For a developer, lack of this knowledge can complicate many activities. For instance, the developer may not know who to ask when questions arise about particular code. For a team lead, lack of this knowledge can make it difficult to know who can bring a new team member up-to-speed in a particular part of the code.

Existing approaches to determining who knows which code have sought to determine who has expertise based on authorship of changes to the code alone (e.g., [11]). These approaches ignore knowledge that is gained by a developer interacting with the code for such purposes as calling the code or trying to understand how the code functions. In this paper, we introduce the degree-of-knowledge (DOK) model that takes a broader perspective on who knows what code by considering both authorship and a developer's interactions with the code. A DOK value for a source code element is a real value specific to a developer; different developers may have different DOK values for the same source code elements. We compute the DOK values for a developer automatically by combining authorship data from the source revision system and interaction data from monitoring the developer's activity in the development environment (Section 4).

To determine whether both authorship and interaction have an effect on knowledge, we gathered data from two professional software development teams. We report on this data to support two claims. First, the code that developers work on changes rapidly. Second, code that developers create and edit overlaps, but is not the same as, the code with which developers interact.

Using this data, we conducted experiments with the members of two development teams to determine the relative effect of authorship and interaction towards modelling knowledge (Section 5). We found that whether or not the developer was the first author of a code element had the most effect on the element's DOK value. However, we also found that all aspects of authorship and interaction improve the quality of the model and help to explain a developer's knowledge of an element.

The availability of DOK models for developers in a team opens up several possibilities to improve a developer's pro-

ICSE '10, May 2-8 2010, Cape Town, South Africa
.

ductivity and quality of work. We consider three possibilities in this paper through exploratory case studies (Section 6). First, we investigate whether DOK values can support finding who is an expert in particular parts of a code base. We found that our approach performed better than existing approaches for this problem that are based on authorship alone. Second, we investigate whether DOK values can help familiarize (onboard) a new team member onto a particular part of the development project. From this study, we learned about kinds of source code for which our current definition of DOK does not adequately reflect a developer's knowledge. Finally, we hypothesized and confirmed that we can accurately identify bug reports that a developer should likely be aware of. We achieve this identification by correlating the developer's DOK values with a bug report's source code changes, even when those changes were made by other team members.

This paper makes three contributions:

- it introduces the degree-of-knowledge model that represents a developer's familiarity with each code element;
- it reports on data about professional developers' authorship and interaction with the code, providing empirical evidence about the rate of information flowing into a developer's environment and the need to consider both authorship and interaction to more accurately reflect the code elements with which a developer is familiar; and
- it reports on the use of DOK values in three different scenarios in professional environments, reporting on the benefits and limitations of the model and demonstrating a measurable improvement for one scenario, finding experts, compared to previous approaches.

2. RELATED WORK

Previous automated approaches to determining the familiarity (expertise) of developers with a codebase rely solely on change information. For instance, the Expertise Recommender [9] and Expertise Browser [11] each use a form of the "Line 10 Rule", which is a heuristic that the person committing changes to a file[1] has expertise in that file. The Expertise Recommender uses this heuristic to present the developer with the most recent expertise for the source file; the Expertise Browser gathers and ranks developers based on changes over time. The Emergent Expertise Locator refines the approach of the Expertise Browser by considering the relationship between files that were changed together when determining expertise [10]. Girba and colleagues consider finer-grained information, equating expertise with the number of lines of code each developer changes [4]. Hattori and colleagues consider changes that have not yet been committed [5]. None of these previous approaches consider the ebb and flow of a developer's expertise in a particular part of the system. The Expertise Recommender considers expertise as a binary function, only one developer at a time has expertise in a file depending on who last changed it. The Expertise Browser and Emergent Expertise Locator represent expertise as a monotonically increasing function; a developer who completely replaces the implementation of an existing method has no impact on the expertise of the developer who originally created the method. Our approach models the ebb and flow of multiple developers changing the same file; a developer's degree-of-knowledge in the file rises when the developer commits changes to the source repository and diminishes when other developers make changes.

The approach we consider in this paper also differs from previous expertise identification approaches by considering not just the code a developer authors and changes, but also code that the developer consults during their work. Schuler and Zimmermann also noted the need to move beyond authorship for determining expertise, suggesting an approach that analyzed the changed code for what code was called (but not changed) [13]. In this way, they were able to create expertise profiles that included data about what APIs a developer may be expert in through their use of those APIs.

In this paper, we go a step further, considering how a developer interacts with the code in a development environment as they produce changes to the code. We build on earlier work from our research group that introduced degree-of-interest (DOI) values to represent which program elements a developer has interacted with significantly [8]. The more frequently and recently a developer has interacted with a particular program element, the higher the DOI value; as a developer moves to work on other program elements, the DOI value of the initial element decays. Our initial applications of this concept computed DOIs across all of a developer's workday [7]. Subsequent work scopes the DOI computation per task [8]. In this paper, we return to the computation of DOI across all of a developer's work to capture a developer's familiarity in the source across tasks.

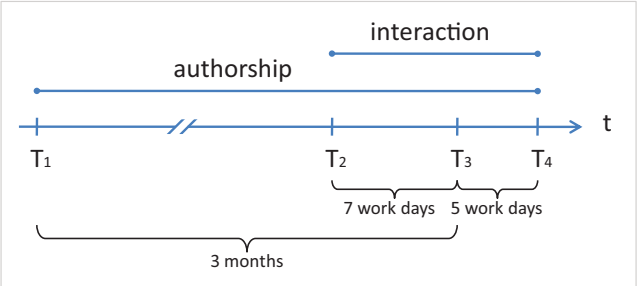
Others have considered the use of interaction data for suggesting where to navigate next in the code [2], for tracking the influence of copied and pasted code [12] and for understanding the differences between novice and expert programmers [14]. None of these previous efforts have considered the use of interaction data for determining expertise in or familiarity with source code.

In a previous study, we considered whether interaction information alone could indicate for which code a developer had knowledge [3]. This study involved nineteen industrial Java programmers. Through this study, we found that DOI values computed from the interaction information can indicate knowledge about a program's source. This study also found that other factors, such as authorship of code, should be used to augment DOI when attempting to gauge a developer's knowledge of the code. This paper builds on this previous work, investigating how a combination of interaction and authorship information indicates a developer's knowledge of code.

3. AUTHORSHIP AND INTERACTION

Existing approaches to representing code familiarity are based solely on authorship (e.g., [11]). Our previous study found that a developer's interaction with the code can indicate the developer's knowledge about source code [3]. These two results suggest a model of code familiarity should be based on both of these factors. However, if there is a strong degree of overlap between the code elements authored and interacted with by a developer, it may be possible to base a model on only one kind of information, as is currently the

[1]We use the term file but many of these techniques also apply at a finer-level of granularity, such as methods or functions.



(a) An Abstract Timeline

	T1	T2	T3	T4
Site1	3/11/2008	22/1/2009	2/2/2009	7/2/2009
Site2	24/11/2008	12/2/2009	23/2/2009	28/2/2009

(b) Specific Points in Time Used at Each Site

Figure 1: Data Collection Time Periods

case with expertise recommenders. To investigate the role of both of these factors, we gathered data from two professional development sites, finding that each presents a unique and valuable perspective on a developer's code knowledge.

$Site_1$ involved seven professional developers (D1 through D7) building a Java client/server system, using IBM's Rational Team Concert[2] system as the source repository. The professional experience of these developers ranged from one to twenty-two years, with a mean experience of 11.6 years (standard deviation of 5.9 years). These developers each worked on multiple streams (branches) of the code; we chose to focus our data collection on a developer's major stream. One developer (D5) could not identify a major stream of the four on which he worked; as this work pattern makes authorship difficult to determine, we have chosen to exclude his data from the presentation given in this section but have included his results in the experiment (Section 5) and case studies (Section 6).

$Site_2$ involved two professional developers, who build open source frameworks for Eclipse as part (but not all) of their work and who use CVS[3] as the source repository system. One developer had three years of professional experience, the other had five years.

Figure 1(a) provides an overview of the different periods of data collection. Authorship information was gathered for a three month period (T_1 to T_3). The interaction data used to compute DOK values was gathered over seven working days (T_2 to T_3). The data reported on in this section is from data collected from T_1 to T_3. The interaction data from T_3 to T_4 was used to update the DOK values as case studies were conducted during this period. Figure 1(b) maps abstract time points to particular dates used for each site.

We report in detail on the data from $Site_1$, providing only an overview of the data from $Site_2$ due to space limitations. Unless otherwise indicated, we report the average (mean) of values with standard deviations (represented as $\pm$).

3.1 Authorship Data

We distinguish between three different kinds of authorship events with respect to a developer D:

- first authorship, representing whether D created the first version of the element,
- number of deliveries, representing subsequent changes after first authorship made to the element by D,
- acceptances, representing changes to the element not completed by D.

We found that the authorship of code loaded into a developer's environment at $Site_1$ changed frequently. At this site, a first authorship, delivery or accept event to an element occurred on average every 54 seconds. The developers had 819 ($\pm$576) first authorships, produced 962 ($\pm$755) delivery events and accepted 153,240 ($\pm 46,572$) changes to an element over three months. The standard deviations for all of these values are high, which is not surprising given the different roles of team members (see Section 7). These aggregate statistics count multiple events happening to the same elements. Considering unique elements, on average, the developers first authored 660 elements[4], delivered to 606 unique elements and accepted changes on 67,437 unique elements. Thus, each day, a developer authored ten new elements, delivered changes to nine elements, and accepted changes to 1068 elements on average over the period T1 to T3.

To provide more insight into this data, we picked a random developer and ten random source code elements that had at least two authorship related events. To give a sense of the ebb and flow of the authorship, we estimated weightings for these events, assigning a first authorship event a value of 1.0, a delivery event a value of 0.5 (since a delivery likely changes just part of the element) and an accept event a negative value of 0.1 (since an accept event corresponds to someone else changing the element). Figure 2 plots the resulting values for each of the ten source code elements over time; each element is represented by a separate line. Only a few of these elements were the target of several events over the three months of data we collected; these elements are indicated by the longer lines in the figure. All elements except one (on the far right) have an accept event after a first authorship or delivery, meaning that someone else on the team has delivered a change to the element; the lines for these elements have a declining slope. Over the three months and the six developers, there is a ratio of 86 to 1 for accept events versus all first authorship and delivery events. This large ratio is indicative of the high rate of change occurring to elements in a developer's environment, caused both by the team members themselves, and by other developers making changes to the team's codebase.

3.2 Interaction Data

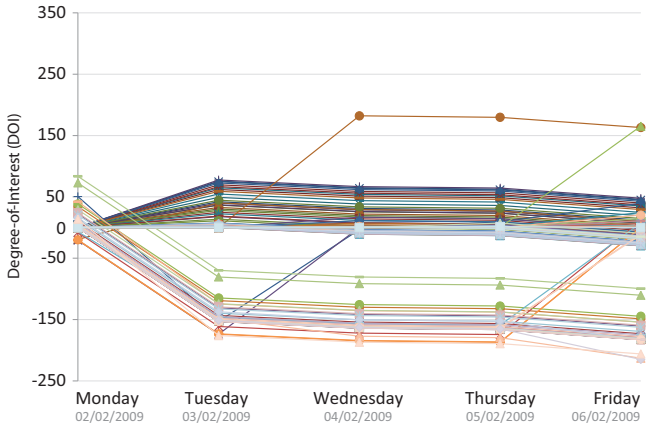
We found at $Site_1$ that developers interacted with many different elements over a week of work, some of them quite frequently.

The developers at this site had an average of 8658 ($\pm$3700) interactions over the seven working days from T_2 to T_3, interacting with 1033 ($\pm$468) distinct elements. As with the authorship information, the difference between individuals is quite substantial as it depends on the individual's role on the team and their individual work patterns. Analyzing the data for the developers separately over the five day period from T_3 to T_4, the number of elements each developer interacted

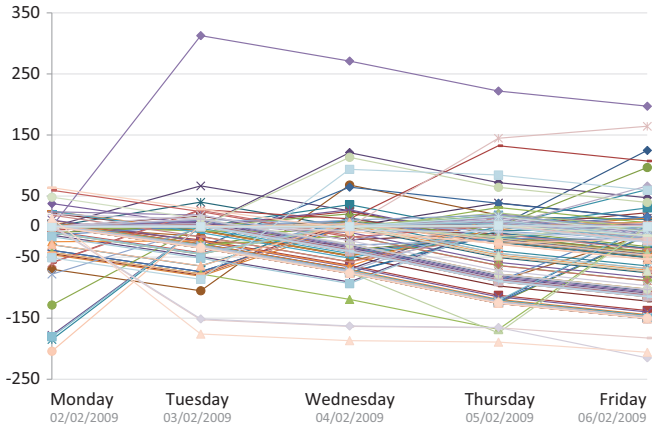
[2] `jazz.net`, verified 01/02/10

[3] `www.nongnu.org/cvs`, verified 01/02/10

[4] The difference between the number of first authorship events and elements first authored is caused by developers merging streams.



(a) D1



(b) D3

Figure 3: Positive DOI Elements

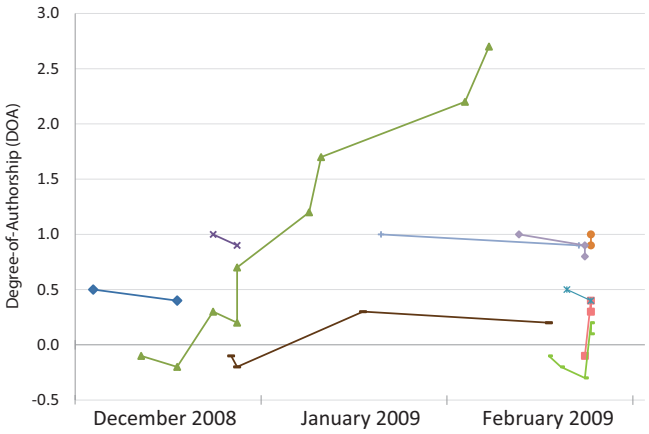


Figure 2: Authorship Events for Ten Elements

with over the prior seven days of interaction is relatively stable at 8258 (±1273).

One way to indicate a developer's ongoing interest in a particular code element is to consider a degree-of-interest (DOI) real value for the element computed from the interaction information. We provide an overview of DOI in the next section (Section 4.2) and DOI is reported in earlier work [8, 6]. A positive DOI value suggests that a developer has been recently and frequently interacting with the element; a negative DOI value indicates a developer has been interacting with other elements substantially since the developer interacted with this element.

At Site_1, on average, each developer had 45 (±6) elements with a positive degree-of-interest per day. We can see this stability in graphs we produced for two developers. Figures 3(a) and 3(b) show, for the period of five working days (T_3 to T_4), elements with a positive DOI value on at least one of the five days for each of the two developers.[5] These graphs show the differences in work patterns across the elements for different developers. Some developers, such as D1 (Figure 3(a)), continuously interact with a group of elements, which results in many lines above zero. Other developers, such as D3 (Figure 3(b)) interact with more elements less frequently, resulting in more lines below zero due to the decay of interest in elements. Most of the six developers also had at least one code element with which he or she was interacting with a lot more than with the rest of the code.

[5]The DOI values shown in these graphs were based on the prior seven days of interaction for each day indicated.

3.3 Authorship and Interaction

We have argued that authorship and interaction can each contribute valuable information to representing a developer's degree-of-knowledge for a source code element. To investigate whether there is a difference in the elements a developer authors versus interacts with, we considered, for each of the five days between T_3 and T_4, the intersection of all code elements that had a positive DOI with all elements that had at least one first authorship or delivery event. On average, out of 45 elements with a positive DOI, only 12 (27%) also had at least one first authorship or delivery event in the previous three months. Thus, interaction information provides a different perspective on the code a developer is working with than solely authorship information.

Our analysis of the data also showed that the number of elements worked with varies with the day of the week. The plot in Figure 4 shows the number of elements with at least one interaction event and the number of elements with a delivery or first authorship event, for each of five consecutive days (T_3 to T_4). While the number of elements developers interacted with holds relatively steady, the number of elements delivered or first authored increases prominently on Friday. The data suggests that the developers *created* changes throughout the week but *delivered* most of them on Friday. This trend suggests that interaction may be a useful predictor of recent source code familiarity whereas authorship helps capture familiarity over a longer period of time.

3.4 Site_2 Data

The data collected at Site_2 shows a lower rate of change in authorship information than Site_1, but even at the lower rate, a substantial amount of the code in a developer's environment is changing each day. The data from Site_2 shows even less overlap between code interacted with than authored.

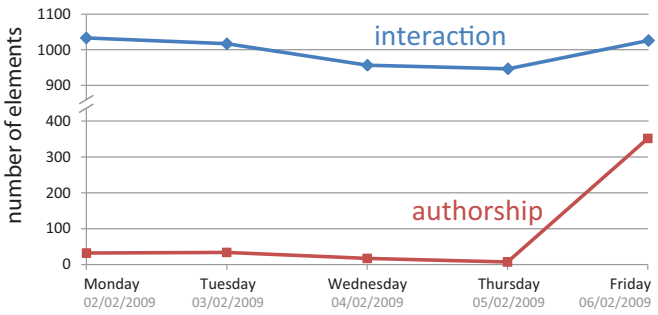


Figure 4: Authorship and Interaction over Five Days

On average, developers at $Site_2$ had a first authorship, delivery or accept event every 700 seconds (compared to 54 seconds at $Site_1$). Considering unique elements and comparing to $Site_1$, the developers authored 2.7 times as many elements (1762 $\pm$ 1835), delivered 2.8 times as many elements (1697 $\pm$ 1746), and accepted changes to only 1/11 as many elements (5977 $\pm$ 3454). The developers at $Site_2$ averaged 6195 interactions over the seven working days, interacting with 566 distinct elements and ending, on average, with 60 elements each day with a positive DOI. The number of elements with a positive DOI that also had at least one first authorship or delivery event is four; an overlap of 7% compared to the 26% overlap at the first site.

There are several potential reasons for these differences. First, whereas the source repository system in use at $Site_1$ supported atomic changesets with explicit accept events occurring within the development environment, at $Site_2$, the source revision system lacked both of these features. Instead, we inferred delivery events based on revision information to source code elements; if a developer performed several commits to the revision system as part of one logical change, we record this as multiple delivery events. Second, the lack of an explicit accept event that could be logged meant that we had to infer at the end of each day that all outstanding changes were accepted, potentially increasing the accept events. Finally, the code at $Site_2$ is smaller and is being worked on by a smaller team, potentially causing a different event profile.

4. DEGREE-OF-KNOWLEDGE MODEL

Our degree-of-knowledge model for a developer assigns a real value to each source code element—class, method, field—for each developer. Our definition of DOK includes one component indicating a developer's longer-term knowledge of a source code element, represented by a degree-of-authorship value, and a second component indicating a developer's shorter-term knowledge, represented by a degree-of-interest value.

4.1 Degree-of-Authorship

From our study of nineteen industrial developers [3], we determined that a developer's knowledge in a source code element depends on whether the developer has authored and contributed code to the element and how many changes not authored by the developer have subsequently occurred. We thus consider the degree-of-authorship (DOA) of a developer in an element to be determined by three factors: first authorship (FA), the number of deliveries (DL) and the number of acceptances (AC).

4.2 Degree-of-Interest

The degree-of-interest (DOI) represents the amount of interaction—selections and edits—a developer has had with a source code element [8]. A selection occurs when a developer touches a code element; for instance, a selection occurs when the developer opens a class to edit the class. An edit occurs when a keystroke is detected in an editor window. The DOI of an element rises with each interaction the developer has with the element and decays as the programmer interacts with other elements. Different kinds of interactions contribute differently to the DOI of an element; for instance, a selection of an element contributes less to DOI than an edit of an element. We use weightings for interactions as defined in the Eclipse Mylyn project, which is successfully supporting hundreds of thousands of Java programmers in their daily work. The Eclipse Mylyn project uses DOI as defined elsewhere [8, 6]. In contrast to Eclipse Mylyn, our use of DOI considers all interaction a developer has with the environment and does not consider any task boundaries indicated by the developer as part of their work.

4.3 Degree-of-Knowledge

We combine the DOA and DOI of a source code element for a developer to provide an indicator of the developer's familiarity in that element. The degree-of-knowledge we compute linearly combines the factors contributing to DOA and the DOI:

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI$$

5. DETERMINING DOK WEIGHTINGS

Completing our definition of a degree-of-knowledge value for a source code element requires determining appropriate weightings for the factors contributing to the degree-of-authorship and for the degree-of-interest. As there is no specific theory we can use to choose the weightings, we conducted an experiment to determine appropriate values empirically. In essence, the experiment involves gathering data about authorship from the revision history of a project, about interest by monitoring developers' interactions with the code as they work on the project and about knowledge by asking developers to rate their level of knowledge of particular code elements. Using the developer ratings, we then apply multiple linear regression to determine appropriate weightings for the various factors.

We report in this section on an initial determination of weighting values based on the data collected from $Site_1$. We then test these weightings at $Site_2$. Our intent is to find weightings that serve as a basis to support exploratory investigations of the degree-of-knowledge model. Determining weightings that might apply across a broader range of development situations would require gathering data from many more projects, which was not warranted at this early stage of investigation.

5.1 Method

At time T_3 in Figure 1, we chose, for each developer, forty random code elements that the developer had either selected or edited at least once in the last seven days ($DOI \neq 0$), or

which the developer had first authored ($FA > 0$) or delivered changes to ($DL > 0$) in the last three months. We chose forty as a compromise between gaining data about enough elements and not encroaching too much on the developer's working time. Each developer was then asked to assess how well he or she knew each of those elements on a scale from one to five. To help the developers with the rating scale, we explained that a five meant that the developer could reproduce the code without looking at it, a three meant that the developer would need to perform some investigations before reproducing the code, and a one meant that the developer had no knowledge of the code. Although this process meant that sometimes we asked the developer about finer-grained code (a field) and sometimes coarser-grained code (a class), subsequent analysis of the data did not show any sensitivity to granularity.

Through this process, we collected 246 ratings for all seven developers. This value is less than the 280 possible ratings because some of the elements we randomly picked were not Java elements (the authorship and interaction data also included XML, Javascript and other types of code) and the developers stated that they would have difficulty rating them; we therefore ignored these elements.

For this experiment, we consider results to be statistically significant with $p<0.05$.

5.2 Analysis and Results

For our first experimental setting, we applied multiple linear regression to the data collected from the source revision logs and the interaction logs collected as the developers from $Site_2$ worked. Multiple linear regression analysis tries to find a linear equation that best predicts the ratings provided by developers for the code elements using the four variables: FA (first authorships), DL (deliveries), AC (accepts) and DOI (degree-of-interest). Multiple linear regression is suitable for our data, even though the user ratings are ordinal, because we are attempting to find an approximation, not a certain class, for the user ratings.

The values of some of the variables, especially DOI and AC can be substantially higher than the values of the other variables. To account for these different scales that could potentially make the weighting factors difficult to ascertain, we applied the analysis both with and without taking the natural logarithms of the values. With the developer rating (on a scale of one to five) as the dependent variable, the best fit of the data was achieved with the values presented in Table 1, when the natural logarithm of the AC and DOI values was used. The resulting DOK equation is as follows.

$$\begin{aligned}\mathbf{DOK} = \quad & 3.293 + 1.098 * \mathbf{FA} + 0.164 * \mathbf{DL} \\ & -0.321 * ln(1 + \mathbf{AC}) + 0.19 * ln(1 + \mathbf{DOI})\end{aligned}$$

Negative values of DOI indicate usage that is not recent. In this analysis, we considered any negative DOI value to be zero so as to not unduly penalize DOK. If we had allowed negative DOI values, then elements that the developer had never interacted with may have a higher DOK value than elements interacted with long ago.

The FA, DL and AC variables are significant in this model and thus help to explain the user ratings. The DOI variable is very close to being significant. An analysis shows that the DOI is not correlated to any of the other variables, suggesting that DOI still plays a predictive role, despite not reaching significance. We hypothesize that the lack of significance is from the lack of elements with a positive DOI in the set of randomly chosen elements. Only 7% of all data points have a positive DOI whereas 28% have a positive FA, 50% have a positive DL and 57% have a positive AC component.

Table 1: Coefficients for Linear Regression

		Weighting	Std. Error	p-value
Intercept	3.293		0.133	<0.001
FA		1.098	0.179	<0.001
DL		0.164	0.053	0.002
$ln(1 + AC)$		-0.321	0.105	0.002
$ln(1 + DOI)$		0.190	0.100	0.059

The F-ratio, a test statistic used for determining the predictive capability of the model as a whole, is 19.6 with $p<0.000001$. This states that the model based on our four predictor variables has a statistically significant ability to predict the user rating. The overall model has an estimated "goodness of fit", R Square, of 0.25 (adjusted R Square is 0.23). R Square represents the fraction of the variation in our user rating that is accounted for by our variables. The correlation coefficient R that represents a measure of the overall fit between our predictor variables and the user rating is 0.50. The standard error of the estimate is 1.17. The 0.25 R Square value shows that our model does not predict the user rating completely. However, the p-value of the overall model as well as the p-values for the variables indicate that there is a statistically significant linear relationship between our model and the user ratings and that each of the four variables contributes to the overall explanation of the user rating.

5.3 External Validity of the Model

To determine if our weightings have any applicability in a different environment, we conducted a similar experiment with the two professional Java developers at $Site_2$. As we did at $Site_1$, we again chose forty random code elements for each developer with the same characteristics as at $Site_1$ and we asked each developer to rank the presented elements from one to five.

We then computed the DOK values for each of the elements using the weightings determined through the earlier experiment with the developers at $Site_1$. To see whether our previously determined model can describe the relationship between the four variables and the developer ratings at $Site_2$, we applied the Spearman rank correlation coefficient statistic. The Spearman rank correlation is a non-parametric statistic that is designed to measure correlations in ordinal data. For the 80 code elements we studied from the two developers there is a statistically significant correlation with $r_s = 0.3847$ ($p=0.0004$). This result suggests that our model can predict DOK values with reasonable accuracy, even when the environments from which the data was collected have different profiles (Section 3.4). We hypothesize that the correlation coefficient is low because the statistic is especially sensitive to individual differences when the sample size is low.

6. CASE STUDIES

To determine if degree-of-knowledge (DOK) values can provide value to software developers, we performed several exploratory case studies. Two case studies were conducted with the seven developers at $Site_1$. A third case study was performed with three developers at $Site_2$; these developers differed from those described in Section 3 as data was collected at a different time to support the study. The three developers were working on a closed-source development effort; one developer was working part-time. The participating developers had an average of 2.5 years of professional experience at the time of the case study.

The first case study considers the problem of finding experts who are knowledgeable about particular parts of the code. The second case study considers a mentoring situation where an experienced developer might use his DOK values to help a new team member become familiar (onboard) into that part of the code base. The third case study considers whether a developer's DOK values can be used to identify which changes to the code might be of interest amongst the many changes occurring during development.

6.1 Finding Experts

The problem of finding experts is to try to identify which team member knows the most about each part of the codebase. Our degree-of-knowledge model applies directly to this problem.

Method.

At $Site_1$, the code is partitioned into projects, where a project is a logical group of Java packages. For this case study, we chose two projects with which most members of the team had interacted. One project comprised 21 Java packages; the other comprised 88 packages. For each class in these packages, and for each of the seven developers participating in our study, we calculated the DOK value for each class-developer pair and then computed DOK values for each package-developer pair by summing the developer's DOK values for each class in the package. Using these values, we produced a diagram, which we call a knowledge map, that showed for each package in a project, the developer with the highest DOK for that package. Figure 5 presents a part of the knowledge map for one project.[6] In this figure, each developer is assigned a colour and each package is coloured (or labelled) according to the developer with the highest DOK values for that package. For one project, 17 of the 21 packages (80%) were labelled and for the second, 61 out of 88 (69%) were labelled. Thus, 78 packages in total were labelled. For the remaining 31 packages, the DOK values for all developers were not positive, meaning primary expertise might lie outside the team.

We then conducted individual sessions with each of the seven team members. In each session, we first showed the developer a list of the packages without any DOK values indicated and asked the developer to write down the name of the team member whom he thought knows the package the best. When requested, we showed the developers a list of the classes within a package. After gathering this data, we showed the developer the knowledge map and asked if the map reflected his view of which developer knows which part of the code. This approach runs the risk of developers over-inflating their expertise in a package to avoid not appearing as an expert in anything. We believe this over-inflation did not happen because the two projects represent only a small fraction of the entire system so a developer who did not indicate expertise in the packages that were part of the case study might still be expert in some other part of the code.

[6]Please note this figure is best viewed in colour.

Results.

We gathered data from six developers (D1-D6); one developer (D7) did not interact with any of the code in the two projects and thus was not able to provide meaningful data.

For the 78 of the 109 packages labelled with a single developer, we gathered 468 (6 developers times 78 packages) assignments from the developers participating in the study. In 301 of these cases (64%), the developers in the study assigned one developer as being the one that "knows the most" or "owns" the package. In 166 of these 301 cases (55%), the result we computed based on DOK values was consistent with the assignments by the developers.

The 55% accuracy value is a lower bound of our approach's performance given that the developer assignments were sometimes guesses; after seeing the knowledge map the developers realized their assignments were likely wrong. All six developers stated that the knowledge map was reasonable, using phrases like it is "close" (D4) and it "reflects [reality] correctly" (D2).

For the 31 out of the 109 packages for which we did not find anyone using the DOK values, the six developers assigned someone to a package in 104 cases. In 48 of these cases (46%), the packages had not been touched for a number of months and were created six months ago. Given that our DOK values were based on three-months of data, we were missing the initial authorship data. Developers stated that in "blank cases" (D4) where our DOK did not determine anyone, we should adapt the DOK to go back further in time.

Comparison to Expertise Recommenders.

For this task, it is possible to compare to other approaches, since earlier work in expertise recommenders has considered the problem of finding experts. As described in Section 2, these approaches are based solely on authorship information. To approximate the results of these earlier approaches, we computed experts for each package by summing up all first authorship and delivery events from the last three months for a developer for each class in the package. Three months was the most history available for these elements due to a major porting of the code at that time. The developer with the most "experience atoms" [11]—the most events—for a package is the expert. We applied this expertise approach to the two projects. In 21 out of the total 109 packages, the expertise approach labelled a package with a different expert developer than our DOK-based approach. For these 21 packages, we had 69 assignments from the six developers. In 34 of these 69 cases (49%), our DOK-based approach agreed with the developer assignments, whereas the expertise approach agreed in only 17 (24%) of these cases. Thus, the DOK approach improves the results for the packages that were labelled differently by the two approaches by more than 100% and the overall result by 11%. This comparison shows that DOK values can improve on existing approaches to finding experts.

Figure 5: Part of a Knowledge Map

6.2 Onboarding

Becoming productive when joining a new development project requires learning the basic structure of the codebase. The process of becoming proficient with a codebase is known as onboarding [1]. In this case study, we investigated whether DOK values computed from developers with experience in a part of a codebase could be used to indicate which code elements a newcomer should focus on when trying to learn that part of the code base.

Method.

For this study, we randomly chose three developers (D1, D3, D5). We asked each developer to describe which code elements from the areas in which he was working would likely be the most useful for a newcomer trying to come up-to-speed on the code. We then generated, for each developer, the twenty elements with the highest DOK and asked the developer to comment on whether these twenty elements would likely be helpful for a newcomer.

Results.

Only 2 of the 60 (3%) elements generated across all three developers were considered by the developers to likely be helpful for a newcomer. The other 58 (97%) elements were described by the developers as not being essential for understanding the code. The elements generated using the DOK values were, "only implementations" (D1) or "secondary consumers" (D3). The developers described that a newcomer only needs to understand basic patterns (D1, D5) and that while the elements generated using DOK could serve as examples, it would be necessary to traverse up the inheritance hierarchy to locate the elements the newcomer should study (D1). These comments are consistent with the description of the developers that they often recommend newcomers to look at API elements. The DOK values for the API elements were either very low or zero as they were neither changing nor were they referred to frequently by the developers who authored them. Perhaps the developers had internalized these APIs and did not need to refer to them.

For onboarding, then, the elements with high DOK values were not considered helpful. However, the developers comments suggest that the elements with high DOK might be used as a starting point to find useful elements for onboarding by following call or type hierarchies. We leave the investigation of this potential use of DOK values to future research.

6.3 Identifying Changes of Interest

In many development projects, keeping up with how the work of teammates might effect one's work typically requires monitoring the progress of changes. In many projects, changes to the source code are tied to bug reports either by listing the bug report in the change or by attaching the change to a bug report (e.g., by attaching the change itself or meta-information such as a task context [8]). By monitoring bug reports instead of inspecting individual source code changes, a developer can be provided more rationale about a change.

Many bug reports for the project, like source code elements, also typically change daily. A developer who wishes to monitor changes of interest must typically perform searches over the bug repository. Determining which changes the developer should consider and ensuring searches are returning interesting changes can be difficult.

In this case study, we investigated whether a developer's DOK values can be used to select changes of interest to the developer because of overlap between the source code change and the developer's DOK model.

Method.

For this study, we computed a DOK model for each of three developers from Site_2.

On a particular day, we determined all bugs that had changed in the previous seven days; we refer to this set as B_C. As these three developers work on multiple projects and we are analyzing changes of interest for one project, we used seven days to capture a sufficient amount of change to the project we were targeting. The same seven days were used for developers S1 and S2; a different seven days were used for developer S3. These dates differed to accommodate developers' schedules.

We then determined the subset of bugs that had change information attached to the bug; we refer to this set as B_{CI}.[7] For each bug report with change information in B_{CI}, we computed the aggregate DOK value for each element in the change information based on the developer's DOK model. We formed the set of bugs with an aggregate DOK value that was positive; we refer to this set as B_{POT}. We then removed the bugs in B_{POT} when the developer was already mentioned on the bug as an assignee, reporter or had commented on the bug. The resulting set of bugs are those we

[7]For this project, the change information were task contexts collected automatically as a developer worked and thus represented both the elements changed in the revision system and elements considered by the programmer in making the change.

Table 2: Size of Bug Recommendation Sets

Developer	B_C	B_{CI}	B_{POT}	B_R
S1	123	26	20	3
S2	123	26	7	2
S3	76	28	5	1

report as bugs of interest to the developer; we refer to this set as B_R. For each bug in B_R, we asked the developer whether they had read the bug or whether they would have wanted to be aware of the bug.

Results.

Table 2 summarizes the number of bugs in each set for each developer. Developer S1 was asked about the relevance of three bugs. He noted that he had read each of these bugs to make sure no further action was required on his part. Developer S2 was asked about two bugs. She noted that for one of these bugs, she was asked in person about the contents of the bug although she had not read it previously. The developer was impressed our approach had picked it out of the many bugs that were undergoing change. Developer S3 was recommended one bug. He had read the bug but did not comment explicitly about whether the bug was relevant to his work.

Overall, our approach provided relevant information to developers in four out of six cases by recommending non-obvious bugs based on the developers' DOK values. While our case study used a very coarse-grained metric to determine relevance of bugs using DOK values, we were able to easily recommend more relevant bugs than noise.

7. DISCUSSION

The degree-of-knowledge (DOK) model is influenced by both the software development process and the software system being developed. We detail a number of the factors influencing DOK and how they pose threats to the validity of the experiments and studies we conducted.

7.1 Amount of Data

Our studies were based on three months of authorship data and seven working days of interaction data. We chose this duration for authorship data based on interviews in our earlier study [3]. In these interviews, developers had suggested three months as a lower bound for the period of time in which one still has knowledge about code after authoring it. Also based on our previous study [3], we chose seven working days of interaction data. Seven working days was the average number of working days that showed a significant result for the correlation between a developer's knowledge and his interaction.

7.2 Multiple Stream Development

The seven developers we studied at the first site share code in streams, which are similar to branches in a source revision system. The developers deliver their changes to one or more streams and accept changes from streams into their workspace. Normally, the developers we studied work only on a small number of streams. However, during our data collection and study period, some of the developers were working on many streams: "it's not a normal situation, right now [it is] very branched out, [and] I almost spend more time merging than working on it" (D5). When streams representing different versions of the same code are merged, additional authorship events are recorded that could skew the results of our experiment and studies. We tried to minimize the influence of these extra events by focusing on only one major stream for each developer.

7.3 Project Phase

Developers interact differently with a codebase depending on the phase of the project on which they are working. In the week in which we collected interaction data at $Site_1$ to determine the DOK weightings, the team was in a testing phase for an upcoming milestone release. Some of the developers reported that they were only performing minor adjustments to the code but not really making any bigger changes to ensure the code did not break. Some developers stated that a couple of months before they were working on new features, during which a substantial amount of new code was created and the focus of individual developers in a part of the codebase was higher.

The number and size of changesets and the tasks of the developers (i.e., testing versus feature development) influences the authorship and interaction data. By taking into account three months of authorship data, seven days of interaction data and also confirming the results of the DOK weighting experiment at a second site, we have tried to minimize the impact of project phases on our results. Further longitudinal studies are needed to better understand the impact of project phases on indicators such as DOK.

7.4 Individual Factors

The first team of seven developers we studied have a strong model of code ownership, with code split amongst team members and certain individuals responsible for certain packages. Other teams we have spoken with have a model of mutual ownership with the team members often working on the same code. The style of code ownership within a team influences the data input to determine DOK values. We have tried to mitigate the risk of these different styles by considering whether the weightings determined for one team applied to another team. However, study of more teams is needed to determine how robust the DOK values are to team and individual styles.

A developer's activity also has an influence on the data. For instance, one developer in our study was working on more than four different streams and was expending effort that week merging streams together. When we applied linear regression on the data points gained from only this developer, the result was not significant. For other developers, the goodness of fit of the model is more than twice as good as the goodness of fit for all developers. Thus, while individual factors, such as the team's style of code ownership and activities of individuals influence results, by having several developers, each with a different activity, we have tried to minimize the risk of individual biases.

7.5 API Elements

In the onboarding case study, API elements affected the outcome: developers suggested API elements as important that DOK values did not capture. The root of the problem is that API elements, by necessity, do not change often. In the three month period we considered for the authorship

component of DOK, there was not a sufficient number of events on the API elements for their DOK to rise based on authorship. Furthermore, as API elements often become basic knowledge, developers do not need to interact with them frequently so the interaction data also does not cause their DOK values to rise. The developers who participated in the case study stated that the elements found using the DOK are often one or two layers below the API elements. A possible improvement to the DOK could be to infer familiarity from subclasses up to the API elements that are the supertypes as it is likely a subclass user knows the API elements to some extent.

8. SUMMARY

On average, six professional Java developers at a site we studied, accepted changes to 1068 source code elements per day and interacted with 923 distinct source code elements over a seven day period. This data confirms what is apparent when one watches a professional developer at work; the amount of information that flows into and changes in a developer's development environment is substantial. By studying professional developers, we also found evidence that the code a developer authors and the code with which the developer interacts are not the same. This high degree of flux and differences in code authored (visible to other team members) from code considered (not visible to other team members) makes it difficult for developers to know who on their team has familiarity with different parts of the codebase.

To help capture and provide access to information about who knows what about the code, we have presented the degree-of-knowledge (DOK) model. By incorporating both authorship and interaction information gathered for a developer, a DOK value for each source code element and developer pair can be produced. We have defined an equation to compute DOK values and set the weightings for the authorship and interaction factors through an experiment with seven professional Java software developers. We confirmed these weightings through a second experiment at a second site with two professional Java developers.

To show that the DOK model can provide value, we report on three exploratory case studies. Through these case studies, we have shown how DOK values can improve upon existing approaches to computing expertise that are based solely on authorship. We have also shown how DOK values might be used to pick out changes of interest in the environment to the developer. Directions for future work in improving DOK were also determined through these studies, such as ways to better capture familiarity in API elements.

The case studies we have conducted provide initial evidence to suggest that further study of the DOK model is warranted. In particular, the weightings for the factors contributing to the DOK model and the appropriate amounts of data to use to compute DOK values require experimentation with more developers in a greater variety of situations.

9. ACKNOWLEDGMENTS

This work was supported in part by IBM and in part by NSERC. We thank all the developers who participated in the experiments and case studies.

10. REFERENCES

[1] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proc. of CHI'07*, pages 557–566, 2007.

[2] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proc. of SoftVis'05*, pages 183–192, 2005.

[3] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proc. of ESEC-FSE'07*, pages 341–350, 2007.

[4] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. of IWPSE'05*, pages 113–122, 2005.

[5] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. *Proc. of MSR'09*, pages 141–150, 2009.

[6] M. Kersten. *Focusing knowledge work with task context*. PhD thesis, University of British Columbia, 2007.

[7] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proc. of AOSD'05*, pages 159–168, 2005.

[8] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of FSE'06*, pages 1–11, 2006.

[9] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proc. of CSCW'00*, pages 231–240, 2000.

[10] S. Minto and G. C. Murphy. Recommending emergent teams. In *Proc. of MSR'07*, page 5, 2007.

[11] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. of ICSE'02*, pages 503–512, 2002.

[12] C. Parnin, C. Görg, and S. Rugaber. Enriching revision history with interactions. In *Proc. of MSR'06*, pages 155–158, 2006.

[13] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proc. of MSR'08*, pages 121–124, 2008.

[14] L. Zou and M. W. Godfrey. Understanding interaction differences between newcomer and expert programmers. In *Proc. of RSSE'08*, pages 26–29, 2008.