Degree-of-Knowledge: Investigating an Indicator for Source Code Authority

Thomas Fritz, Jingwen Ou and Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
{fritz,jingweno,murphy}@cs.ubc.ca

ABSTRACT

Working on the source code as part of a large team productively requires a delicate balance. Optimally, a developer might like to thoroughly assess each change to the source code entering their development environment lest the change introduce a fault. In reality, a developer is faced with thousands of changes to source code elements entering their environment each day, forcing the developer to make choices about how often and to what depth to assess changes. In this paper, we investigate an approach to help a developer make these choices by providing an indicator of the authority with which a change has been made. We refer to our indicator of source code authority as a degree-of-knowledge (DOK), a real value that can be computed automatically for each source code element and each developer. The computation of DOK is based on authorship data from the source revision history of the project and on interaction data collected as a developer works. We present data collected from eight professional software developers to demonstrate the rate of information flow faced by developers. We also report on two experiments we conducted involving nine professional software developers to set the weightings of authorship and interaction for the DOK computation. To show the potential usefulness of the indicator, we report on three case studies. These studies considered the use of the indicator to help find experts, to help with onboarding and to help with assessing information in changesets.

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

1. INTRODUCTION

A common experience in today's information rich societies is the feeling of being deluged by information. For instance, querying the web in your favourite search engine for the keywords "software engineering research" returns tens of millions of results; a Google web search on March 14, 2009 returned approximately 39,300,000 results. A human filtering through these results considers implicit and explicit cues about the authority and quality of the pages returned. For authority, the human most often relies on the link structure of the web [10], which corresponds to the rank of the page returned. For quality, the human uses both explicit—a biography of the page's author—and implicit—the organization and layout of the site containing the page—cues [1].

Software developers also face a deluge of information daily. The integrated development environments developers use provide fast access to the many (often millions of) lines of code comprising the system on which they work. The information available through the development environment is not static, but is constantly being changed by members of the team. Similar to the web search case, a developer must thus also constantly assess the authority and quality of information available within the environment. For instance, when receiving a group of source revisions—a changeset—from a team member that is intended to solve a bug or implement an enhancement, a developer must assess the impact of the changeset: will the revisions integrate easily or will they cause the system to break? In contrast to the web search case in which there are a variety of cues available to the human to differentiate the pages returned in the search, a developer typically has only a few implicit cues. For instance, a developer may use implicit knowledge about who authored the changeset: revisions from a known long-time team member with a track record for quality may be accepted without question whereas revisions from a new team member may be examined more closely.

Throughout a day of programming, a developer is constantly faced with such assessments. For one group of professional developers we studied, each developer, on average, accepted changes to over one thousand source code elements per day over a three month period (Section 4). Correct assessments enable a productive workday. Incorrect assessments result in misplaced or wasted effort, with consequences such as the introduction of faults when reviews or other crosschecks on the codebase were not performed.

We believe that providing explicit, reliable cues about the authority and quality of information entering a developer's environment can enable developers to direct their available effort towards the parts of the development more likely to cause problems. In this paper, we investigate an explicit indicator for source code authority. The idea is that if we could transmit an indicator of authority with new information, such as a changeset, entering a developer's environment, the receiving developer could make an assessment about the quality of the information more accurately and quickly.

As an indicator for authority about source code elements, we introduce a degree-of-knowledge (DOK) value that can be automatically computed from data gathered about and during the development process (Section 3). 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 by combining authorship data from the source revision system and interaction data from monitoring the developer's activity in the development environment.

To determine how authorship and interaction combine to represent a developer's knowledge of a source code element, we conducted experiments with two groups of professional developers (Section 5). We found that the factor with the most effect on a source code element's DOK value for a developer was whether or not that developer was the first author of the element. 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.

We also report on three case studies we conducted about the use of DOK values to indicate authority in source code (Section 6). One case study uses DOK values to consider which developers on a team are knowledgeable (have authority) in different areas of the codebase. We found that our approach performed better than existing approaches based on authorship alone. The second case study uses DOK values to help with onboarding a new team member onto a development project. From this study, we learned about kinds of source code for which our current definition of DOK does not adequately indicate authority. The last case study considers the use of DOK values to help with changeset assessment. Our approach produced promising results for identifying elements that would help with understanding, and presumably, assessing elements when receiving a changeset.

This paper makes the following contributions:

- it introduces and empirically investigates an explicit indicator for authority, a degree-of-knowledge (DOK) value, that incorporates both a developer's authorship of code elements and a developer's interaction with code elements.

- 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 works.

- it reports on the use of DOK values in three different scenarios in an industrial environment, reporting on the benefits and limitations of our indicator and demonstrating a measurable improvement for one scenario, finding experts, compared to previous approaches.

2. RELATED WORK

Earlier work has considered how to determine which developers are experts in particular parts of a source codebase (e.g., [13]). These papers use expertise in the sense of having "the skill of an expert" [13]. In this paper, we have deliberately chosen to use the term authority because, rather than trying to identify individuals with a certain skill level in a part of the code base, we want to capture whether a developer altering code is "an accepted source of expert information or advice"[1] in the part of the codebase that is changing.

Most previous automated approaches to determining software development expertise rely on change information available from a project's source repository. For instance, the Expertise Recommender [11] and Expertise Browser [13] each use a form of the "Line 10 Rule", which is a heuristic that the person committing changes to a file[2] 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 in quantifying experience, and hence, expertise [12]. Girba and colleagues consider finer-grained information, equating expertise with the number of lines of code each developer changes [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). 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 [9]. The more frequently and recently a developer has interacted with a par-

[1]`www.thefreedictionary.com/authority`

[2]Please note that we use the term file but many of these techniques also apply at a finer-level of granularity, such as methods or functions.

ticular 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 [8] and filtered the views in the environment based on DOI values. However, this approach did not sufficiently scope the display of appropriate elements. Subsequent work scopes the DOI computation per task [9]. In this paper, we return to the computation of DOI across all of a developer's work to capture a developer's authority in the source across tasks.

Others have considered the use of interaction data for suggesting where to navigate next in the code [3], for tracking the influence of copied and pasted code [14] and for understanding the differences between novice and expert programmers [15]. None of these previous efforts have considered the use of interaction data for determining expertise or authority in source code.

In a previous study, we considered whether interaction information alone could indicate for which code a developer had knowledge, or in the terms of this paper, authority [4]. 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 impacts a developer's knowledge of code.

3. AN INDICATOR FOR AUTHORITY

We refer to the indicator for source code authority that we investigate in this paper as a degree-of-knowledge (DOK), which is a real value, computed per developer and assigned to a source code element. We use the term source code element to refer to a class (type), method, or field in a source codebase. 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.

3.1 Degree-of-Authorship

From our study of nineteen industrial developers [4], we determined that the 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 D_1 in an element to be determined by three factors:

- first authorship (FA), representing whether D_1 created the first version of the element,
- the number of deliveries (DL), representing subsequent changes after first authorship made to the element by D_1,
- acceptances (AC), representing changes to the element not completed by D_1.

3.2 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 [9]. 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, selections of an element contribute less to DOI than edits of an element. We use the DOI as defined in the Eclipse Mylyn project, which is successfully supporting hundreds of thousands of Java programmers in their daily work. Details on the DOI computation can be found elsewhere [9, 7].

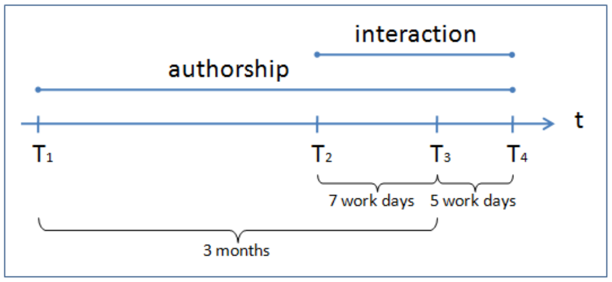


Figure 1: Timeline.

3.3 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 authority in that element. The degree of knowledge we compute linearly combines the factors contributing to DOA and the DOI:

$$DOK = \alpha_1 * FA + \alpha_2 * DL + \alpha_3 * AC + \beta * DOI$$

To determine appropriate weightings ($\alpha_1, \alpha_2, \alpha_3, \beta$), we conducted an experiment with professional Java developers; this experiment is described later in the paper (Section 5).

4. DATA OVERVIEW

For the experiments and studies we report on in this paper, we collected data from two sites ($Site_1$ and $Site_2$). Figure 1 provides an overview of the data collection. Authorship information for the developers involved in the study at each site was gathered for the entire time period (T_1 through T_4), resulting in almost three and a half months of authorship data. We gathered a total of twelve days of interaction information (T_2 to T_4).

$Site_1$ involved seven professional developers building a Java client/server system. The professional experience of these developers ranged from one to twenty-two years, with an average experience of 11.6 years (± 5.9 years). These developers each worked on multiple streams (branches) of the code. Most developers focused their work on one stream, even though they were working on several streams; we chose to focus our data collection on a developer's major stream. One developer (D5) could not identify a major stream, but worked more equally on more than four streams; this work pattern makes authorship difficult to determine and we exclude his data from the presentation given in this section.[3] Table 1 summarizes the time periods over which data was collected for this site.

[3]The data from developer D5 was included in the experiment (Section 5) and case studies (Section 6).

	T_1	T_2	T_3	T_4
$Site_1$	3/11/2008	22/01/2009	2/02/2009	7/02/2009
$Site_2$	24/11/2008	12/02/2009	23/02/2009	28/02/2009

Table 1: Data Collection Periods for each Site.

$Site_2$ involved two professional developers, who in part of their time, build open source frameworks and tools for Eclipse. We analyzed data only for the open source portion of their work. These developers had an average of four years of professional experience. Table 1 summarizes the time periods over which data was collected for this site.

To provide an overview of the data we use in this paper, we use various statistics to characterize both the authorship and interaction data. For space reasons, we report in detail only on data from $Site_1$. A summary of the data for $Site_2$ is provided at the end of this section.

4.1 Authorship Data

At $Site_1$, on average, a first authorship, delivery or accept event on an element occurred every 54 seconds. The developers first authored 819 ($\pm$576) elements, produced 962 ($\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 authored 660 elements, delivered to 606 unique elements and accepted changes on 67,437 unique elements. Over the period T_1 to T_3, a developer thus, on average, authored ten new elements, delivered changes to nine elements and accepted changes to 1068 elements, each day.

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 assigned a first authorship event a value of one, 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 resultant values for each of the ten source code elements over time. Only a few of the elements in the plot are the target of several events over the three months of data we collected. All elements except one have an accept event after a first authorship or delivery, meaning that someone else on the team has delivered a change to the same element; the lines representing 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 problem of overwhelming change we described in the introduction.

4.2 Interaction Data

The developers had an average of 7861 ($\pm$3982) interactions over the seven working days from T_2 to T_3, interacting with 923 ($\pm$518) 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 interacted with over the prior seven days of interaction is relatively stable at 7500

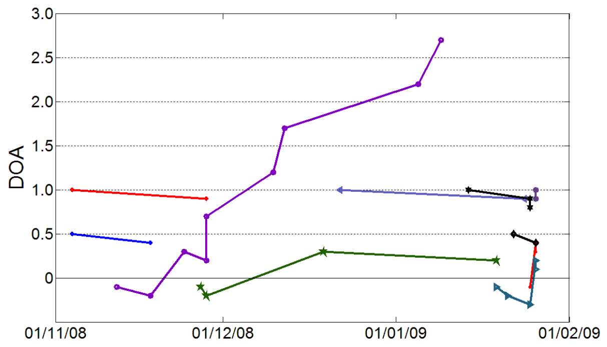


Figure 2: Authorship Events for Ten Elements.

($\pm$1135).

On average, each developer had 43 ($\pm$7) elements with a positive DOI per day. We can see this relative 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 two developers.[4] These graphs show the differences in work patterns across the elements for different developers. Some developers, such as D1 (Figure 3(a)) interact with a large group of elements. For other developers, their interaction varies a lot between the elements and only smaller groups of elements have a similar interaction pattern as can be seen for D3 (Figure 3(b)). 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.

4.3 Authorship and Interaction

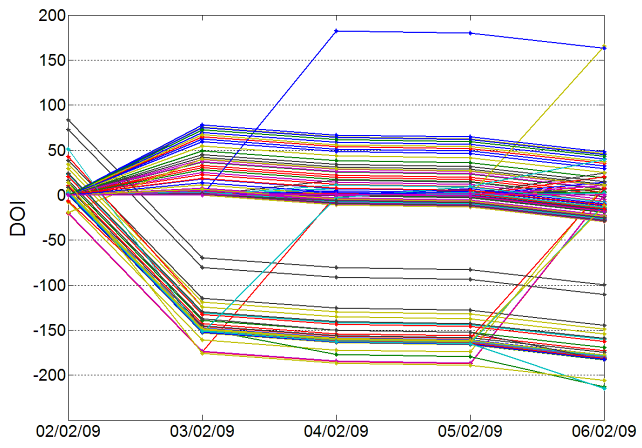
Our assumption is that a degree of knowledge indicator should include components for authorship and for interaction. To gain insight into whether both of these components are important and cover different aspects of development, we looked, for each of the five days between T_3 and T_4, at the intersection of all elements that had a positive DOI with all elements that had at least one first authorship or delivery event. On average, out of 43 elements with a positive DOI, only 11 (26%) also had at least one first authorship or delivery event over the last three months.

To give a sense of the stability in the authorship and interaction data, we plotted, for five consecutive days (T_3 to T_4), the number of elements with at least one interaction event during the last 7 working days and the number of elements with a delivery or first authorship event over the last three months. The values are relatively stable except for that the number of elements developers interacted with decreases towards Thursday (5/2/09), and the number of elements delivered to the stream increases prominently on Friday (6/2/09). This data reflects that the developers were quite active on Monday, created changes throughout the whole work week but delivered most of them on Friday.

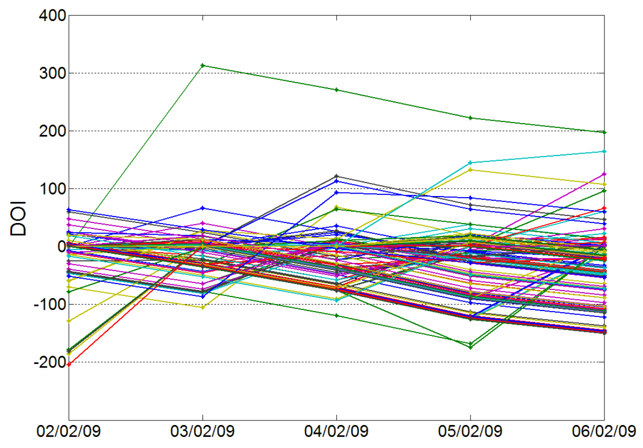
4.4 $Site_2$ Data

On average, developers at $Site_2$ had a first authorship, delivery or accept event only every 700 seconds (compared to 54 seconds at $Site_1$). Considering unique elements and

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



(a) Developer 1



(b) Developer 3

Figure 3: Positive DOI Elements.

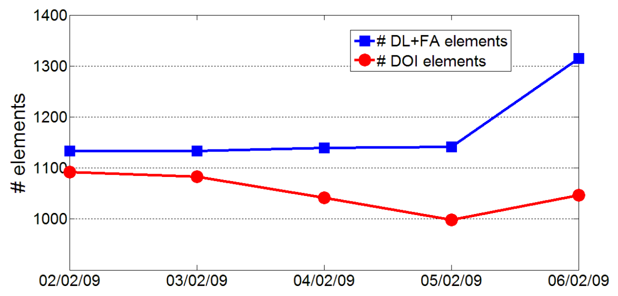


Figure 4: Overall Elements for DOA and DOI.

comparing to $Site_1$, the developers authored, on average, 2.7 times as many elements (1762 ± 1835), delivered 2.8 times as many elements (1697 ± 1835), and accepted changes to only 1/11 as many elements (5977 ± 3454).

The developers at $Site_2$ had an average of 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. These numbers are fairly similar to $Site_1$. However, the number of elements with a positive DOI that also had at least one first authorship or delivery event is only four which results in 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 in the development environment, at $Site_2$, the source revision system lacked both of these elements of support. Instead, we inferred delivery events based on revision information to source code elements; if a developer committed several times 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 codebase at $Site_2$ is smaller and is being worked on by a smaller team, potentially causing a different event profile.

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 on which we can 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 codebase as they work on the project and about knowledge by asking developers to rate their level of knowledge in 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 used these weightings to test their applicability at $Site_2$.

5.1 Method

At time T_3 in Figure 1, we chose, for each developer, fourty random code elements that the developer had either selected or edited at least once in the last seven days, or which the developer had authored or delivered in the last three months. 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 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.

This process resulted in 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 and the developers stated that they would have difficulty rating them; we therefore ignored these elements.

5.2 Results

For our first experimental setting, we applied multiple linear regression to the data collected from the source revision logs and the logs collected as the developers 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 independent 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 1 to 5) as the dependent variable, the best fit of the data was achieved with the values presented in Table 2, when the natural logarithm of the AC and DOI values was used. In this analysis, we also chose to consider only positive values of the DOI variable. Negative values of DOI indicate usage that is non-recent usage and we do not want to penalize our indicator in reducing knowledge beyond that of an element which has not had any interactions and whose DOI value would be zero.

	Coefficient	Standard Error	p-value
Intercept	3.293	0.133	2.32E-68
FA	1.098	0.179	3.16E-09
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

Table 2: Coefficients for Linear Regression.

The FA, DL and AC independent variables are significant[5] in this model and thus help to explain the user ratings. The DOI variable is very close to being significant. We hypothesize that this 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 compared to 28%, 50% and 57% for FA, DL and AC respectively.

The F-ratio, a test statistic used for determining the predictive capability of the model as a whole, is 19.6 (p = 5.59E-14). 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 independent 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 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 independent variables indicate that there is a statistically significant linear relationship between our model and the user ratings and that each of our four variables contribute to the overall explanation of the user rating.

5.3 Testing the Weightings

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 fourty 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 independent variables and the developer ratings at $Site_2$, we applied the Spearman rank correlation coefficient statistic. The Spearman rank correlation is a non-parametric rank statistic that is designed to handle ordinal data and is robust to the distribution. For the 80 code elements we studied from the two developers there is a statistically significant (p=0.0004) correlation with $r_s = 0.3847$. This result shows that our model can be applied in a different environment as an indicator for source code authority, even when the sources of the data have different profiles (Section 4.4). We hypothesize that the correlation coefficient is not high because the individual differences in rating elements do not randomize sufficiently with only two developers.

6. CASE STUDIES

To determine if degree-of-knowledge (DOK) values can provide value to software developers, we performed three case studies with the seven developers at $Site_1$. These case studies were conducted in the five working days after the weighting experiment (T_3 to T_4 in Figure 1). The first case study considers the problem of finding experts 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 onboard into that part of the code base. The third case study considers the changeset scenario we introduced to motivate the paper.

6.1 Finding Experts

The problem of finding experts is to try to identify which team member knows most about each part of the codebase. By ranking the results of individual developer's authority in a codebase using DOK, we can apply our approach to this problem.

Method.

We chose two projects[6] with which most members of the team had interacted. One project comprised 21 Java packages; the other one 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. We then produced a diagram, which we call a knowledge map, that showed for each package, the developer with the highest DOK for that package. A part of the knowledge map for one project is presented in Figure 5.[7] For the first project, 17 of the 21 packages (80%) were labelled with a single developer. For the second project, 61 out of 88 (69%) were labelled with a

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

[6]A project is a logical group of Java packages.

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

single developer. Thus, 78 packages in total were labelled with a single developer.

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 she 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.

Results.

We gathered data from six developers; 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" (D2, D5) or "owns" (D1, D6) the package.[8] 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 not always correct: the developers stated that they were sometimes guessing and that after seeing the knowledge map 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 earlier, 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 authorship and delivery events from the last three months for a developer for each class in the package. The developer with the most "experience atoms" [13]—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-base approach. For these 21 packages, we had 69 assignments from the six developers. In 34 of these 69 cases (49%), our DOK-based approach was correct, whereas the expertise approach was correct in only

[8]Developers used these words interchangeably.

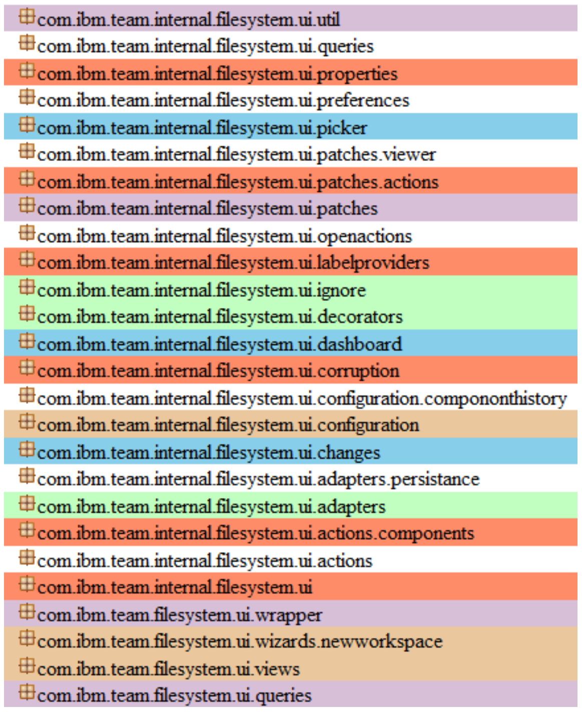

com.ibm.team.internal.filesystem.ui.util
com.ibm.team.internal.filesystem.ui.queries
com.ibm.team.internal.filesystem.ui.properties
com.ibm.team.internal.filesystem.ui.preferences
com.ibm.team.internal.filesystem.ui.picker
com.ibm.team.internal.filesystem.ui.patches.viewer
com.ibm.team.internal.filesystem.ui.patches.actions
com.ibm.team.internal.filesystem.ui.patches
com.ibm.team.internal.filesystem.ui.openactions
com.ibm.team.internal.filesystem.ui.labelproviders
com.ibm.team.internal.filesystem.ui.ignore
com.ibm.team.internal.filesystem.ui.decorators
com.ibm.team.internal.filesystem.ui.dashboard
com.ibm.team.internal.filesystem.ui.corruption
com.ibm.team.internal.filesystem.ui.configuration.compononthistory
com.ibm.team.internal.filesystem.ui.configuration
com.ibm.team.internal.filesystem.ui.changes
com.ibm.team.internal.filesystem.ui.adapters.persistance
com.ibm.team.internal.filesystem.ui.adapters
com.ibm.team.internal.filesystem.ui.actions.components
com.ibm.team.internal.filesystem.ui.actions
com.ibm.team.internal.filesystem.ui
com.ibm.team.filesystem.ui.wrapper
com.ibm.team.filesystem.ui.wizards.newworkspace
com.ibm.team.filesystem.ui.views
com.ibm.team.filesystem.ui.queries

Figure 5: Part of a Knowledge Map

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.

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 [2]. 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 the code base. For instance, DOK values might be used to filter the display of code elements in a project to focus on those with which the newcomer's mentor has familiarity.

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. We showed the developer the top twenty elements according to the developer's DOK values and asked her to comment on whether these twenty elements were of use for a newcomer.

Results.

Only 2 of the 60 (3%) elements generated across all three developers were labelled by the developers as likely useful for someone new to the area of code. 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, for the interesting elements it is necessary to traverse up the inheritance hierarchy (D1). The developers described that they often recommended elements that were part of the API. The DOK values for the API elements were either very low or zero as they were neither changed nor were they referred to frequently by the developers who authored them.

For onboarding, the direct DOK values are not 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 Changeset Assessment

As outlined in the introduction to this paper, a developer who receives a changeset from another team member must assess the changeset before accepting it into his development environment. We wanted to determine if it is possible to ship with a changeset, additional elements that represent the developer's authority in that change so that a receiving developer could better assess the incoming change set.

Method.

Overall, our method for this case study involved taking a changeset, selecting additional source code elements that might provide authority for that changeset based on DOK values from the developer creating the changeset, and asking the creating developer to rank which additional elements are useful for understanding the changeset. We chose this method because unless we are able to produce elements that the creator thinks help understand the changeset, it is unlikely that a receiving developer would view the additionally provided elements with a changeset as adding authority for the revisions.

The case study involved four developers (D1, D2, D3, D6). For each of these developers, we randomly selected one changeset that the developer had delivered within that week. On average, the four changesets considered comprised four classes, nine methods and 72.5 lines of code. To determine the additional elements, we computed the structural context of the changeset, consisting of all methods and fields referencing or referenced by a changed method and all direct supertypes and subtypes for a changed type. We computed DOK values for each element in the structural context and selected as the additional elements the twenty elements with the highest DOK values.

We then asked each developer to look at the changeset they had created and describe which other elements in their development environment are useful to understand the changeset. Then, we presented the twenty elements with high DOK values that we had selected and asked the developer to rate their usefulness for understanding the changeset on a scale from one (not useful) to five (very useful). We also asked each developer to comment on the differences between the elements they indicated as useful and the high DOK list for their changeset.

Results.

Overall, the developers rated 33% of the additional elements selected with a rating of four or five, stating that these elements are useful and "essential" (D6) to understand the changeset. Most elements that the developers suggested as being useful and that were not yet covered by the high DOK elements were API elements.

On the surface, a 33% accuracy rating suggests that DOK is not applicable to this problem. However, given the coarseness of the input data and the naive approach we took in generating the elements, the result does hold some promise. When we generated the structural context from which we selected elements with high DOK, we based the generation of the context on all statements in changed methods, and not just the changed statements. As on average 61% of the statements in the methods affected by the changeset remained the same, the resultant structural context was not focused on the change data and thus was very coarse when selecting the high DOK values. Second, the changesets considered in this study contained changes from more than one task, such as refactorings and changes to method visibility, further polluting the selection of authoritative elements. Finally, as we noted in the onboarding section, it may be possible to include API elements by following links from elements with high DOK; however, this might requires heuristics to identify API elements. In light of these issues, having 33% of generated elements rated highly suggests the approach does have promise if more care is taken in the element selection process.

7. DISCUSSION

The degree-of-knowledge (DOK) indicator of source code authority we study in this paper 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 [4]. 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 [4], 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 both the onboarding and the changeset assessment case studies, 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 definition, do not change often. In the three month period we considered for the authorship component of DOK, there were not sufficient 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 studies stated that the elements found using the DOK are often secondary users that are one or two layers below the API elements. A possible improvement to the DOK could be to infer authority 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. CONCLUSION

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. With changes to over one thousand source code elements per day, a developer does not have time to fully assess all changes before accepting them into their environment. Indiscriminate acceptances can cause faults in the code (breaks of builds) that require the developer to expend time and effort to fix.

In other environments where humans must assess an overwhelming amount of information, indicators of authority of the information are used. For instance, the authority of a book published on a subject by an expert may be considered to be higher than a recently added wikipedia entry where the authorship of the information is harder to trace. As another example, humans have come to rely on the authority of web pages as computed by graph-based link measures; pages with higher ranks based on these measures often appear higher in results produced by web search engines.

To help developers better assess information flowing into their development environments, we have presented an indicator for source code authority called a degree-of-knowledge (DOK). 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.

We also reported on three case studies aimed at a preliminary investigation of the use of DOK as an indicator for authority. Through these case studies, we showed how DOK values can improve upon existing approaches to computing expertise that are based solely on authorship. We also showed that there is promise for using DOK to help a developer assess the authority of a set of revisions entering their environment. Directions for future work in improving DOK as an indicator for source code authority were also determined through these studies, such as ways to better capture authority in API elements.

Many efforts in software engineering research are aimed at increasing the amount of information provided to software developers. As just one example, awareness techniques (e.g., [6]) are intended to provide developers pre-warnings of potential conflicts with changes by providing some level of additional information into the development environment. In this paper, we have considered the other side of the coin:

how can a developer better cope with the overwhelming amount of information already entering their environments? Future study in this area should consider the combination of source code analysis and DOK values to better represent a developer's authority in the code, indicators for authority of other kinds of information, and studies to better understand how authority might help a developer assess information available to them.

9. ACKNOWLEDGMENTS

This work was supported by the IBM Ottawa Center for Advanced Studies and NSERC. We thank all the developers who participated in the experiments and case studies.

10. REFERENCES

[1] B. Amento, L. Terveen, and W. Hill. Does "authority" mean quality? predicting expert quality ratings of web documents. In *Proc. of the 23rd Annual Int'l ACM SIGIR Conf. on Research and Development in Info. Retrieval*, pages 296–303. ACM, 2000.

[2] 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 the SIGCHI Conf. on Human Factors in Comp. Systems*, pages 557–566. ACM, 2007.

[3] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proc. of the 2005 ACM Symp. on Soft. Vis.*, pages 183–192. ACM, 2005.

[4] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proc. of the the 6th Joint Meeting of the European Soft. Eng. Conf. and the ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 341–350. ACM, 2007.

[5] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. of the Eighth Int'l Workshop on Principles of Soft. Evol.*, pages 113–122. IEEE Computer Society, 2005.

[6] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. In *Proc. of the 2004 ACM Conf. on Computer Supported Cooperative wWrk*, pages 21–24. ACM, 2004.

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

[8] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proc. of the 4th Int'l Conf. on Aspect-oriented Software Development*, pages 159–168. ACM, 2005.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Soft. Eng.*, pages 1–11. ACM, 2006.

[10] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[11] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proc. of the 2000 ACM Conf. on Computer Supported Cooperative Work*, pages 231–240. ACM, 2000.

[12] S. Minto and G. C. Murphy. Recommending emergent teams. In *Proc. of the Fourth Int'l Workshop on Mining Software Repositories*, page 5. IEEE Computer Society, 2007.

[13] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. of the 24th Int'l Conf. on Soft. Eng.*, pages 503–512. ACM, 2002.

[14] C. Parnin, C. Görg, and S. Rugaber. Enriching revision history with interactions. In *Proc. of the 2006 Int'l Workshop on Mining Software Repositories*, pages 155–158. ACM, 2006.

[15] L. Zou and M. W. Godfrey. Understanding interaction differences between newcomer and expert programmers. In *Proc. of the 2008 Int'l Workshop on Recommendation Systems for Soft. Eng.*, pages 26–29. ACM, 2008.