Expert Recommendation with Usage Expertise

David Ma[1] davma@ucalgary.ca
David Schuler [2] ds@cs.uni-sb.de
Thomas Zimmermann[1,3] tz@acm.org
Jonathan Sillito [1] sillito@ucalgary.ca

[1] Department of Computer Science, University of Calgary, Canada
[2] Department of Computer Science, Saarland University, Germany
[3] Microsoft Research, USA

Abstract

Global and distributed software development increases the need to find and connect developers with relevant expertise. Existing recommendation systems typically model expertise based on file changes (implementation expertise). While these approaches have shown success, they require a substantial recorded history of development for a project. Previously, we have proposed the concept of usage expertise*, i.e., expertise manifested through the act of calling (using) a method. In this paper, we assess the viability of this concept by evaluating expert recommendations for the ASPECTJ and ECLIPSE projects. We find that both usage and implementation expertise have comparable levels of accuracy, which suggests that usage expertise may be used as a substitute measure. We also find a notable overlap of method calls across both projects, which suggests that usage expertise can be leveraged to recommend experts from different projects and thus for projects with little or no history.*

1. Introduction

For software development teams, a question often asked is *who knows what?* Developers, struggling with source code, may pose such questions as *who can help me with this file?* Managers, charged with the responsibilities of assigning tasks or organizing code reviews, may ask *who has experience with SQL?* Ideally the best candidate for these tasks would be the person who can produce the best result in the shortest time; in other words, those who have the *most relevant expertise*. Inconveniently these experts may often be located on different floors or even in different locales. Considering that developers separated by only 20 meters communicate as infrequently as developers separated geographically [1], it is unlikely that the developers will know who the experts are.

Early research into this area involved the use of expertise recommendation systems based on *manually provided expertise data*. However, the data in these systems was rarely precise and rarely up to date [9]. Recent approaches will often automatically infer developer expertise based variants of the *Line 10 Rule*. The Line 10 Rule stems from a version control system that stores the commit author in line 10 of the log message. Through the act of changing a file, the developer is considered to have gained expertise for the said file. This type of expertise is also referred to as **implementation expertise** [2]. While such approaches have shown promise, the implicit criteria for recommending developers for a file is that there must be at least one commit by the developer for the file. Thus these recommendation systems cannot be applied to files/projects with no or little history.

In earlier work we proposed **usage expertise** [13]; the accumulation of expertise by *calling* (using) methods. By simply calling a method, a developer demonstrates that they know what the method does (without necessarily knowing implementation details). Furthermore, when method calls are added to existing code, developers are required to have an understanding of the source code surrounding the location of change (usually characterized by existing method calls). In other words, developers demonstrate usage expertise for the added method call *and* the surrounding **context**.

In this paper, we compare the accuracy of usage expertise based recommendations (including context) against implementation based recommendations through an empirical study of the ECLIPSE and ASPECTJ projects. We also propose an approach towards leveraging usage expertise to recommend developers from projects that use the same external libraries or frameworks (*recommending across projects*). For example, is it possible to recommend developers from the ECLIPSE project for tasks in the ASPECTJ project (or vice versa)?

Our results indicate that usage expertise, with context, can recommend with accuracy similar to implementation expertise. This suggests that usage expertise can be as a substitute for implementation expertise based systems. Furthermore, results indicate that providing cross-project recommendations is possible given an adequate overlap of

shared external dependancies. The implications of this are that not only can usage expertise produce recommendations across projects, but also, that it may be possible to recommend for projects with no or little history.

In Section 2 we describe how we use reconstructed CVS commits to model developer expertise (expertise profiles). Section 3 discusses the set of heuristics we use to infer developer expertise from developer expertise profiles. Section 4 details how experiments have been applied in this study. Section 5 presents the corresponding results of our experiments. Section 6 outlines the potential threats to validity. Section 7 summarizes related research. Section 8 concludes the paper with a summarization of our findings as well as a discussion of future avenues of study.

2. Quantifying Expertise

2.1. Data Collection

Our approach can be applied to any version control system history. However, we base our data collection on CVS repositories since many open source projects currently make use of it.

We reconstruct a CVS history via the use of the APFEL tool [15] which clusters file revisions into a CVS commit using a *sliding time window approach* [16]. A reconstructed operation is composed of one or more file revisions r. Each revision is the result of a single CVS commit R, where $r \in R$. Each commit can then be aggregated into a history of CVS commits H, where $R \in H$.

2.2. Activity Computation

By reconstructing commit operation we can then compute the changes made during each commit. These changes are computed using a comparison of the abstract syntax tree of a file both before and after a commit R.

We can compute the methods that have been added or changed within a commit operation R (implementation expertise), denoted as $C(R)$. We define $C(R)$ as follows:

$$C(R) = \{m \mid r \in R, m \in D(r)\}$$

- Where m is a method
- Where $D(r)$ is the set of added or changed methods for a revision r.

In the example below, the commit R_{ex} yields the following set of changed methods:

$$C(R_{ex}) = \{\texttt{update}()\}$$

Additionally we compute the **multi-set** (i.e., a set allowing for duplicate members) of added method calls $N(R)$ during a commit (usage expertise). We define $N(R)$ as follows:

$$N(R) = \{n \mid m \in C(R), n \in E(m)\}$$

- Where n is a method call
- Where $E(m)$ is the set of added method calls for a changed method m

In the example below, the commit R_{ex} added calls to three methods within the method body of `update()`:

$$N(R_{ex}) = \{\texttt{addTest}(), \texttt{worked}(), \texttt{refreshStatus}()\}$$

The set of new method calls $N(R)$, the set of added or changed methods $C(R)$ and the author of a commit $author(R)$ serve as the input for the construction of expertise profiles.

2.3. Expertise Profiles

For each project developer the data mined from the version archives is aggregated into an *expertise profile*. Such a profile contains the methods changed by a developer (implementation), the methods called (usage) as well as the frequency of each change or call. We can define the expertise profile P for a developer d as follows:

$$P_d = (I_d, U_d, cfreq_d, ufreq_d)$$

I_d is the set of methods changed by a developer d over a history of CVS commits H. U_d denotes the set of methods called by a developer d over H. We now define I_d and U_d as follows (m denotes a method):

$$I_d = \{m \mid m \in C(R), R \in H, author(R) = d\}$$

$$U_d = \{n \mid n \in N(R), R \in H, author(R) = d\}$$

Let $cfreq_d$ be the frequency in which a method is changed. $cfreq_d$ is defined as follows:

$$cfreq_d(m) = \ \mid \{R \mid R \in H, m \in C(R), author(R) = d\} \mid$$

Similarly let $ufreq_d$ be the frequency in which a method is called. Also recall that $N(R)$ is a multi-set which allows for more than one instance of a method call:

$$ufreq_d(m) = \ \mid \{R \mid R \in H, n \in N(R), author(R) = d\} \mid$$

As an example, Table 1 shows the expertise profile of Erich Gamma mined from the ECLIPSE CVS archive. For brevity, we report only simple method names. Erich mostly changed methods related to testing and UI (which is no surprise because he is one of the inventors of JUNIT), and used listeners and progress monitors frequently. We note that it is also possible to compute expertise profiles on a weekly (or monthly) basis to model only recent developer activity.

Table 1. Partial expertise profile for E.Gamma

Six most frequently changed		Six most frequently used	
`createPartControl`	185	`addSelectionListener`	72
`aboutToStart`	163	`openError`	57
`createControl`	148	`addModifyListener`	35
`rerunTest`	143	`refreshStatus`	31
`menuAboutToShow`	142	`addTest`	29
`testFailed`	136	`worked`	26

2.4. Partial Program Analysis

Unlike Williams and Hollingsworth [14], our approach does not build (compile, link) *snapshots* of a system to compute inserted method calls. As they point out, such interactions with the build environment (compilers, make files) are extremely difficult to handle and will inevitably result in high computational costs. Instead, we analyze only the differences between single revisions (commit log comparisons). Our preprocessing is then fundamentally cheaper as well as platform/compiler independent. Note that method signatures cannot be fully resolved. Instead we are limited to identifying methods using only their names (e.g., `addTest`) and the number of arguments (e.g., (1)). As a result, the precision of call identification is roughly 68% [15]. However, alternative approaches to partial program analysis can raise precision to as much as 91% [4]. For more details on how changes are recovered from commits, we refer to the APFEL plug-in [15].

3. Scoring Expertise

We propose six different heuristics which infer expertise (usage or implementation) by analyzing developer activity. Given a query Q and a developer d: each heuristic will quantify a developer's expertise denoted as $E_d(Q)$. Each heuristic produces a ranked ordering of the most qualified developer(s).

3.1. Usage Expertise Heuristics

Here we infer developer aptitude with heuristics based on *usage profiles*. We propose four possible measures because we do not know yet which heuristic yields the "best" recommendations and to gain insight into what characterizes an effective measure. In the context of inferring usage expertise we define the contents of each query Q as a non empty set of *method calls*.

3.1.1 Depth of Method Knowledge

Each call for a method is quantified as a linear increase in expertise for the method. Thus for a set of methods the developer with the most expertise is the developer with the largest sum of calls. Formally we define this as:

$$E_d(Q) = \sum_{m \in Q} ufreq_d(m)$$

Given a set of methods Q, expertise is defined as the sum of the frequency of calls for any method m where $m \in Q$.

The computation of expertise is based on situations where developers will utilize a similar set of method calls. A developer will have comparatively more expertise over the queried set if he/she has demonstrated a greater depth of understanding over a set of methods (judged by frequency).

3.1.2 Breadth of Method Knowledge

Here we consider developers to have expertise for a method with at least a single call. Thus the best developer(s) for a queried set of methods are the developers who have called the most members of the set at least once. Formally we define this as:

$$E_d(Q) =\mid \{m \mid m \in U_d, m \in Q\} \mid$$

Thus given a set of methods Q we define the expertise of a developer d in terms of how many methods m, where $m \in Q$, were called at least once.

An example situation suited for this heuristic can be described using two developers Alice and Bob who develop similar components of a system; thus implying similar method call patterns and frequencies. Alice, has demonstrated knowledge over a larger set of methods (that also happen to belong in the query set Q) and thus is determined to have comparatively more expertise than Bob.

3.1.3 Relative Depth of Method Knowledge

Frequency of calls by a single developer are normalized relative to frequency of calls by the entire developer population. Commonly used methods are weighed less than rarely called methods. Formally we define this as:

$$E_d(Q) = \sum_{m \in Q} \frac{ufreq_d(m)}{ufreq_D(m)}$$

Where D is the entire developer population and $ufreq_D$ (m) is the frequency in which m was invoked by all developers.

Consider for instance a Q that contains `extract` (called 1000 times) and `insert` (called 500 times). Again we will return to our favorite developers Alice and Bob. Alice invokes the `extract` method a total of 15 times and `insert`

method 5 times. Meanwhile Bob invokes both `extract` and `insert` a total of 10 times each.

$$Alice = \frac{15}{1000} + \frac{5}{500} = 0.025$$

$$Bob = \frac{10}{1000} + \frac{10}{500} = 0.03$$

Each developer has an equivalent number of overall method calls but under this measure Bob is said to have more knowledge of the methods in set Q.

3.1.4 Relative Breadth of Method Knowledge

Again we consider developers calling methods at least once to have knowledge. However, calls to methods used by a large portion of developers is scored lower than methods called by only a few developers. This measure favors developers who have called the widest range of methods with an emphasis on rarely called methods.

This measure combines both approaches where expertise is quantified using 1) the breadth of expertise over Q and 2) the frequency in which a method was called *relative* to the global frequency. Let $c_d(m)$ be a function with the range of $\{0, 1\}$: 1 indicating that a developer has called a method at least once, 0 otherwise. We formally define $c_d(m)$ as:

$$c_d(m) = min(\mid \{m \mid m \in U_d, m \in Q\} \mid, 1)$$

Given $c_d(m)$ we now quantify expertise as follows:

$$E_d(Q) = \sum_{m \in Q} \frac{c_d(m)}{\sum_{d \in D} c_d(m)}$$

Where $\sum_{d \in D} c_d(m)$ is the number of developers for a project that have called method m at least once.

3.2. Implementation Expertise Heuristics

Using the *implementation profile* of a developer and a query Q in the form of a set of methods, we infer the developer's aptitude for the query as described below.

3.2.1 Method Change Frequency

Anvik and Murphy [2] measure implementation expertise using frequency of changes to *files*. Conversely, we measure implementation expertise using frequency of changes to *methods*. A developer who commits changes to methods is said to have acquired knowledge for that set. Higher change frequencies will then imply a greater depth of knowledge for a particular method, that is:

$$E_d(Q) = \sum_{m \in Q} cfreq_d(m)$$

Recall that $cfreq_d$ is defined as the frequency of methods changes. Thus for a given set of methods Q, expertise for a developer d is measured as the raw frequency of changes done on methods belonging to Q.

It may be the case that developer changes are local to a particular location. But the raw frequency of changes made to that location may be large enough such that a developer who has made a disproportionate amount of changes to a single location have more inferred expertise (under this heuristic) than a developer who has made changes over a broader spectrum of methods.

3.2.2 Most Recent Change(s)

Fritz and Murphy [5] suggest that it is not only the frequency of developer activity but also the recency of activity as indicators of knowledge. As an increasing amount of change is applied to a method it is often the case that the structure and behavior of a method will deviate further and further from previous incarnations. This measure quantifies implementation expertise on the basis of developers who have demonstrated the most *recent* expertise over Q. The time in which a developer d has changed a method m can be obtained by the numeric timestamp associated with a commit R.

Thus the developer who has the highest sum of timestamps is the developer who has most recently changed the set of methods Q. We now define this measure as follows:

$$E_d(Q) = \sum_{m \in Q} T_d(m)$$

Where $T_d(m)$ is the numeric timestamp representing the most recent date in which developer d modified method m. $T_d(m)$ is 0 when developers have never modified m.

Note that this heuristic favors developers who have made the most recent changes to the overall set Q rather than individual elements of the set. Consider the following example where $Q = a, b$:

$T_{Alice}(a) = 100$	$T_{Alice}(b) = 0$	$E_{Alice}(Q) = 100$
$T_{Bob}(a) = 75$	$T_{Bob}(b) = 70$	$E_{Bob}(Q) = 150$
$T_{Carol}(a) = 90$	$T_{Carol}(b) = 90$	$E_{Carol}(Q) = 180$

Although Alice has committed a more recent change than Bob or Carol she is classified as having smaller degree of expertise given the smaller scope of her changes.

4. Methodology

In evaluating usage expertise, we considered our objectives in terms of two dimensions:

- Is usage expertise as precise as implementation expertise when recommending developers for a single project?

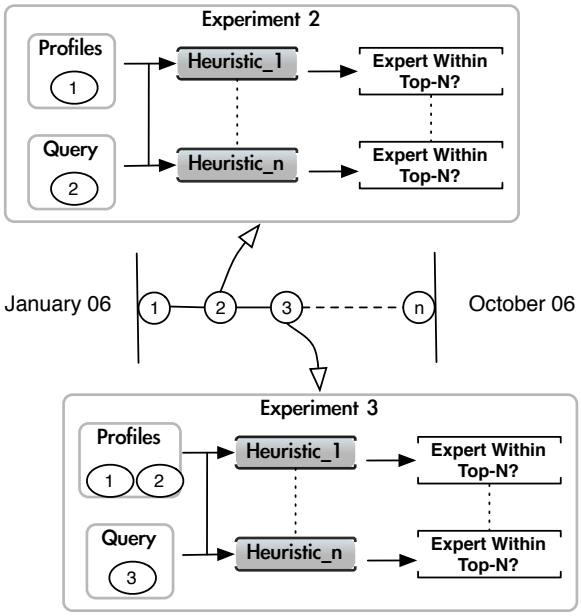

Figure 1. Experiment Overview

- Could usage expertise be used to recommend developers outside of a particular project (across projects)?

We base our evaluation on data mined from the ECLIPSE and ASPECTJ CVS repositories. Specifically, we take data occurring between the months of January and October of 2006 (roughly ten months) to form profiles/evaluate expertise. The choice of a ten month data collection period is not arbitrarily chosen, but rather the reasons are threefold:

- Because of previous experience with the data collected during this time.
- The variance in team size and the corresponding commit rates. ECLIPSE, having almost ten times the number of active committers when compared to ASPECTJ, recorded roughly nine times the number of commits during this ten month period (9000 versus 1000 commits). We are confident that anomalies associated with individual commits will be mitigated with the sheer mass of evidence (i.e., the 1000 commits).
- By limiting the period of time in which data is collected we implicitly reduce the impact of *knowledge decay* as a confounding factor. Constraining the historical reach of profile data reduces instances where developers have all but forgotten a method. Conversely should a developer continually demonstrate activity on a particular method (call or change) then the developer is intimately aware of the method and thus be safely considered an expert.

4.1. General Experiment Procedure

Figure 1 depicts the general procedure used for our experiments. Given a commit R_m, we create a developer expertise profile for each developer in a project, P_d, trained on repository data **prior to** R_m. We then apply the previously described heuristics using P_d to quantify the amount of expertise each developer has for a set Q. Thus each heuristic will assess a score for each developer, of which the highest score(s) indicate the developers with the most expertise. We then incrementally update P_d using the recovered changes in R_m and repeat the above procedure for all remaining commit operations ($R_{m+1} \ldots R_n$). We determine the precision of each heuristic in terms of the overall percentage in which a 'successful' recommendation appears in the Top-N results (hit rate).

We apply this procedure to two different sets of experiments. The details of each experiment, how Q is formed and how we define success are explained below.

4.2. Recommending Within Projects

These experiments compare the precision of usage and implementation expertise in the context of providing recommendations for individual projects. To perform this comparison, we test with three different types of queries:

Implementation. The *added or changed methods* during a commit R

Usage. The *added method calls* during a commit R.

Usage with context. The *added method calls* during a commit R plus the method calls *added prior to* R for the same location (context).

We define a successful recommendation as the appearance of the actual expert for Q within the Top-Nth recommendation. Where the commit author is considered to be the actual expert. The underlying assumption is that whoever made (committed) the change had the expertise to do so. In some cases we may recommend a developer who may have had the expertise but did not do the change. However, accounting for such a situation would lead to higher hit rates. In other words, the precision reported in this paper can be considered a lower bound.

4.3. Recommending Across Projects

This set of experiments is intended to determine the possibility of making cross project recommendations with usage expertise. In other words, is it possible to recommend developers working on the ECLIPSE project for tasks in ASPECTJ (and vice versa).

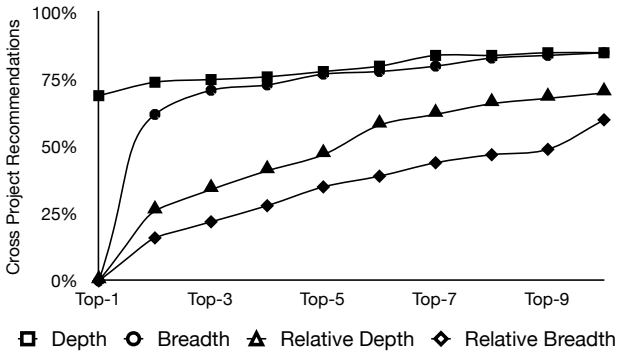

Figure 2. Recommending Across Projects

To test this we now consider commits from both projects. For any given commit we train usage profiles on data, belonging to either project, occurring prior to the commit. We limit our possible queries to only the *method calls added* during a commit (i.e., the **Usage** scenario from above).

Here we define a successful recommendation as a developer recommended from other projects. That is, given a Q created from a reconstructed ECLIPSE commit, a successful recommendation would be a developer from the ASPECTJ project in the Top-Nth recommendation (and vice versa). Note that without a user study we cannot be certain if these recommendations correct; we leave this as future work.

5. Findings

5.1. Expertise Within Projects

Figure 3 illustrates our results for recommending within projects. The left plot shows favorable results when recommending for ASPECTJ. With the first recommendation (N=1), 3 of the 4 usage based measures perform close to the best implementation based heuristic. As N increases, hit rates begin to converge until finally usage overtakes implementation when N is 7. Note that ASPECTJ had roughly 20 active committers and thus as N increases experts may incidentally appear in the Top-N rankings without strong evidence of developer expertise.

However, this trend does not repeat for ECLIPSE project (right plot). When N is less than 9, the accuracy of usage based heuristics is less than the accuracy of implementation based heuristics. We speculate that the disparity in results could possibly be attributed to the scope of dependencies on external libraries. With a smaller code base, modules in ASPECTJ are likely to be managed by only a small set of individuals. ECLIPSE on the other hand has multiple authors for any given sub-project which in turn increases the probability that a file has been touched by multiple developers. Thus these two activity trends may result in a smaller overlap used methods in ASPECTJ (more likely to recommend the commit author) and a larger overlap of used methods in ECLIPSE (less likely to recommend the commit author).

5.2. Context Improves Accuracy

Figure 4 depicts the increased hit rates when recommending with usage plus context based queries. The improvement is especially pronounced in the ECLIPSE project with roughly a 25% improvement over hit rates from using only usage expertise queries. Given as few as the Top-2 recommendations we see that Change Frequency is roughly 75% accurate (compared to under 50% without context). This suggests that (unsurprisingly) only a handful of developers contribute the bulk of changes to methods. So it is likely that developers understand the calls surrounding the location of change, given that they were directly responsible for adding the previous methods; evidence that querying with context is based on solid ground.

5.3. On Usage Expertise Heuristics

Results do not clearly depict any one heuristic as superior. Thus we cannot yet conclude how to best leverage usage expertise for recommendation. Conversely, it may also be the case that perhaps inferring based on breadth or relative depth of expertise are indeed equally effective.

What we can conclude is when considering the commit author as the only expert, inferring with *Depth* yields lower hit rates. This disparity is especially pronounced when recommending for the ECLIPSE project using both usage and usage with context. Similar to our explanation in Section 5.1, ECLIPSE developers may rely on commonly used methods. When developers (other than the commit author) call these common methods frequently they will be extension score higher in expertise, thus explaining these results. This is not to say that *Depth* is inherently a poor measure, but rather, that it recommends developers with the most expertise in commonly used methods. *Relative Depth* on the other hand favors developer activity consisting of calls to rarely called methods; methods that the commit author is likely to be the sole user of. Thus results lead us to conclude that *Relative Depth* accurately models the behavior of the commit author and will yield higher results when the commit author is considered as the expert.

5.4. On Recommending Across Projects

Figure 2 paints a favorable picture regarding the prospects of recommending across projects. Intuitively, for the hit rate to be as high as 75%, there must be sufficient overlap in the calls to external libraries. By extension this

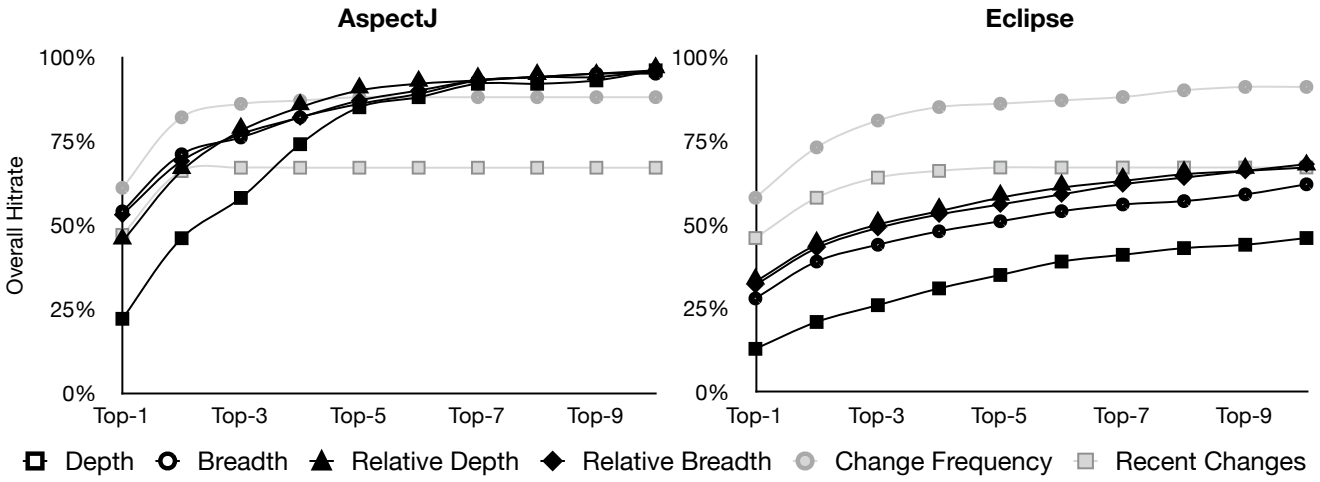

Figure 3. Hit rates within projects. Implementation (light) versus usage expertise (dark).

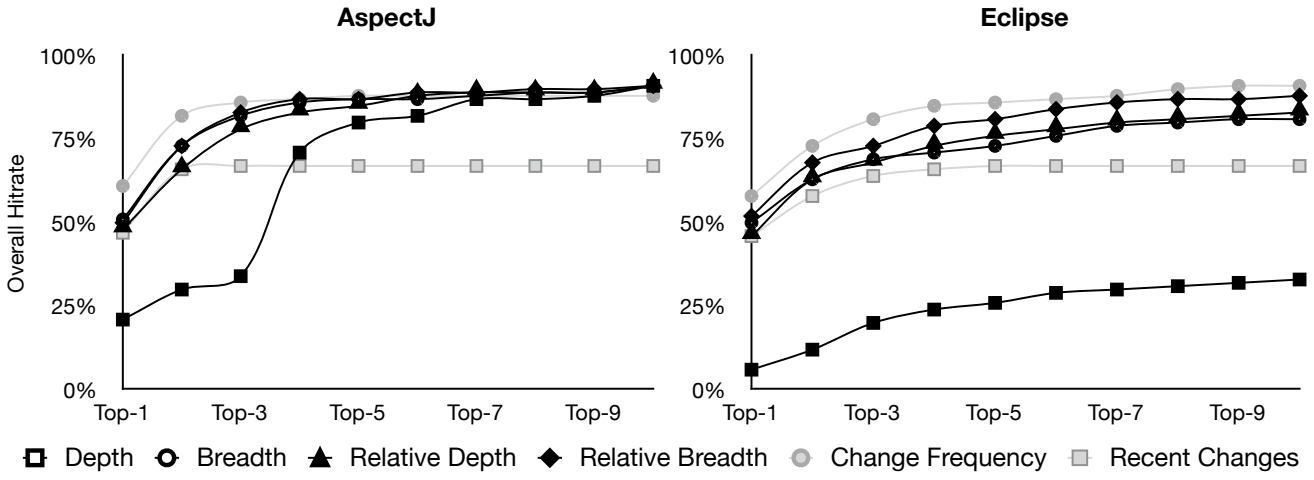

Figure 4. Hit rates within projects. Implementation (light) versus usage expertise with context (dark).

means that recommendations are at the very least, possible. When rarely called methods are given more weight (*Relative Depth/Breadth*), we are more likely to recommend inter-project developers, given that the overlap of internal methods is likely to be low.

While we cannot claim for our experiment that the cross-project recommendations are in fact "good" recommendations, we demonstrated the feasibility of providing recommendations across projects. Assessing the quality of these recommendations is only possible with user studies; which we leave as future work.

6. Threats to Validity

The ability to generalize our results to a larger set of projects is bound by the limited size of our data set. Our experiments were applied to two projects and although a common trend was observed in our results we cannot claim that these patterns apply to other projects. Further, both projects are open source which entails its very own distributed development structures. We do not claim that our observations hold when considering proprietary software projects.

With respect to evaluating the effectiveness of recommendations across projects we note that the pool of developers in each evaluated project is mutually exclusive. Thus it was the case that recommendations could not be made across projects using implementation expertise. We con-

sider this a limitation rather than a threat to the external validity of our results. Certainly there are instances where developers are involved in multiple projects. But it is seldom the case where multiple developers contribute to the same set of projects. Thus it is challenging to test without mining a contrived data set for expertise profiles.

Measuring the accuracy of heuristics was done via a comparison of the list of recommendations with the author of a commit operation. Given the level of indirection between us and the development, team we cannot say for sure if the author is in fact responsible for all the changes during a commit. Further, it is difficult to determine if the author is truly an expert without having direct contact with the author him or herself. However, we argue that this is not a confounding factor. It is may be the case that the commit author did not implement the changes but still acquires experience through reviewing of such code (in accordance to commit policies). Also, the author may not be the developer with the most expertise but certainly he or she has demonstrated that they have sufficient expertise to commit a change.

Our approach (with exception of the Most Recent Change heuristic) does not account for both the implicit and explicit decay of programmer knowledge as time progresses. As previously mentioned we limit our analysis of data to commits occurring within the span of under a year as one possible technique mitigate the effects of knowledge decay by discarding knowledge that is years old. Still the presence of false positive recommendations exists due to heuristics which do not account for such decay. Recent work by Matter [8] incorporates a 3% weekly decay on inactive developer activities provides the best level of precision and recall. While the details of their experiments differ than the ones described in this paper the model of a weekly decay of knowledge can also be applied to our heuristics.

By considering only recorded CVS commits we also acknowledge the potential for false negatives. That is, developers that may in reality have expertise over a set of methods but is not accounted for in our definition of an expertise profile. Developers may have accumulated such through activities external to the modification of code (documentation, involvement with design process, communicating with peers) or perhaps in other projects.

7. Related Work

Our approach for expert recommendation is not the first to mine software repositories. However, previous approaches are based on variants of implementation expertise, implying that they are specific to a given project. By mining usage expertise instead, we get *project independent* expertise which is transferable *across projects*. This allows us to recommend for newcomers to software projects. In addition our approach allows for *recommendations for code with little or no history*.

Anvik and Murphy propose and compare three different ways to determine implementation expertise from bug reports [2]. Two approaches involve analyzing source repository check-in logs. Here a bug report is linked to the source repository to obtain a change set for each report. Then the containing module for each entry in the change set is determined at file or package level. From this containing module a list of all developers who previously made changes to it is compiled. In their third approach, they determine expertise from a bug network. A bug network consists of all bugs that are connected by a relationship (e.g. duplicate, depends on). From this network they use the carbon-copy lists, comments and resolver information to compile a list of experts.

Minto and Murphy present the Emergent Expertise Locator (EEL) that presents experts to a developer based on the recently edited or selected files [11]. Their approach uses the coordination requirements framework introduced by Cataldo et al. [3]. The experts for a set of files is computed by using information from the version control system on which files are changed together and how often a developer changed a particular file.

Mockus and Herbsleb present the Expertise Browser [12] that uses experience atoms (EA) as a measure of expertise. These EA's are gathered by mining the source repository using author and change information. Then each EA can be associated with several domains (such as author, organization, technology or release version). Later the experience atoms can be queried for finding experts for the different domains.

McDonald and Ackerman propose a recommendation architecture called Expertise Recommender [10] that uses expertise profiles for organization members. These profiles are built using two heuristics: change history and calls to tech support. The change history heuristic assigns expertise to all authors that modified a file and the tech support heuristic assigns expertise based on previously completed support calls. When a new call to tech support comes in, these profiles are queried to find members of the organization that can assist in solving the problem at hand.

Recently Kagdi, Hammad and Maletic propose a tool, xFinder [6], that ranks expertise based primarily upon the changes to files. Specifically candidates were ranked along the premises of change expertise, experience, as well as the proportion of contributions of a developer. Similar to the approach taken in this paper, their data collection is also computationally lightweight given that their data can be obtained by only examining commit logs. Using as many as eight open source projects they conclude that the range of accuracy of their recommendation falls between 43% and 82%. However, xFinder requires projects to have a decent amount of recorded history and also cannot recommend across projects.

8. Conclusions and Consequences

In this paper we showed empirically that usage expertise produces recommendations with an accuracy comparable to implementation expertise. We also presented an approach to improve recommendations by also considering the implicit understanding of the surrounding context.

Our results also revealed that between the ECLIPSE and ASPECTJ project there is a substantial overlap of calls to external (or shared) libraries; alluding to the possibility of cross-project recommendations. While we did not assess the correctness of cross-project expert recommendations, we have demonstrated the possibility in this paper. We expect that the precision will be similar across projects to the precision within projects, for which usage expertise performs similar to implementation expertise.

To summarize, usage expertise can enhance traditional expert recommendation systems. In particular, they will allow recommendations for files and projects with little to no history and from unrelated parts of a project.

In addition to the work presented in this paper, we plan to expand on the following in future work:

- **Recommending for projects with little to no history**. We plan to conduct an empirical study to determine if recommendations can be made for brand new projects or for projects without an existing history.
- **Evaluating precision across projects**. We plan to conduct a user study to determine if cross-project recommendations are precise.
- **Relation between usage expertise and quality**. Experts produce code of higher quality (e.g., fewer defects) than novices. An empirical study is needed to determine if experts, as judged by heuristics inferring usage expertise, also produce a corresponding level of quality?
- **Improve precision of data collection**. In our study, method calls are not fully resolved because we use a conservative strategy for ambiguities like overwritten and overloaded methods. We plan to use *fragment class analysis* [4] to collect precise data on archived repositories. Further, we will also mine real time method usage data via an ECLIPSE plugin.
- **Combining implementation and usage expertise**. A combined usage and implementation expertise metric may also lead more precise results. In addition, taking both types of expertise into account can be used to assign different roles for developers. For instance they can be considered as consumers or producers of methods, which can help to facilitate communication amongst them.
- **Combining expertise with tasks and bug reports**. Additional data sources such as task and bug reports can augment profiles. By monitoring tasks, we can recommend tasks to developers (which might be relevant for their work). Bug reports can also be enriched with expertise information, e.g., when a bug report contains source code or stack traces, we can recommend experts for it.
- **Measuring API consumption**. For API producers, we can provide information about how their API is used and *by who*. They can ask consumers for feedback on the API, inform them about upcoming changes, or even inform them about serious bugs in the API. Moreover, they can obtain information about the popularity of methods. This helps to prioritize and plan efforts where heavily used methods would receive more attention, while rarely used methods are candidates for deprecation.

9. Acknowledgements

A shorter version of this paper will be published at ICSM 2009 [7]. Many thanks to the ICSM review committee and Reid Holmes for valuable feedback on this project. This research was funded in part by NSERC and an IBM Jazz Faculty grant.

References

[1] T. J. Allen. *"Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information Within the RD Organization"*. MIT Press, 1977.

[2] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

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

[4] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 313–328, New York, NY, USA, 2008. ACM.

[5] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM.

[6] H. H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *ICSM*, pages 157–166. IEEE, 2008.

[7] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, September 2009.

[8] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR 09: Proceedings of the 2009 intl. working conference on Mining software repositories*. ACM, 2009.

[9] D. W. McDonald and M. S. Ackerman. Just talk to me: a field study of expertise location. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 315–324, New York, NY, USA, 1998. ACM.

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

[11] S. Minto and G. C. Murphy. Recommending emergent teams. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, page 5, 2007.

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

[13] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of the Fifth International Working Conference on Mining Software Repositories*, May 2008.

[14] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[15] T. Zimmermann. Fine-grained processing of CVS archives with apfel. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, New York, NY, USA, October 2006. ACM Press.

[16] T. Zimmermann and P. Weissgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.