Mining Usage Expertise from Version Archives

David Schuler
Department of Computer Science
Saarland University
Saarbrücken, Germany
ds@cs.uni-sb.de

Thomas Zimmermann
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
tz@acm.org

ABSTRACT

In software development, there is an increasing need to find and connect developers with relevant expertise. Existing expertise recommendation systems are mostly based on variations of the *Line 10 Rule*: developers who changed a file most often have the most implementation expertise. In this paper, we introduce the concept of *usage expertise*, which manifests itself whenever developers are using functionality, e.g., by calling API methods. We present preliminary results for the ECLIPSE project that demonstrate that our technique allows to recommend experts for files with no or little history, identify developers with similar expertise, and measure the usage of API methods.

Categories and Subject Descriptors: D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—*Version control*;

General Terms: Human Factors, Management

1. INTRODUCTION

More and more software is developed across multiple sites in large geographically distributed teams. As teams get larger and structures become more complex, finding developers best suited for a specific task gets more difficult [3]. Ideally, this would be the developer that has the *most expertise* in the techniques and skills required for this task, and thus, would produce the best result in the best possible time.

Organizations use different ways to track their members' expertise. One of the main responsibilities of a team manager is to know people's expertise to allocate the development resources in the most economic way. While this does work for smaller collocated teams, it does not scale to large projects that are developed across multiple sites. Often systems that use *manually provided expertise* data are introduced to assist in expertise location. These systems, however, are rarely precise and kept up-to-date [5]. Thus, there is a need for systems that infer expertise for developers automatically, for instance by leveraging data mined from version archives.

Most automatic approaches use variants of the *Line 10 Rule* to determine experts for files. The name Line 10 Rule stems from a version control system that stores the author who did the commit, in line 10 of the commit's log message. Developers who changed a file are considered to have expertise for this file. This heuristic, however, does not work for files and developers with no or little history. To recommend a developer for a file, there has to be at least one commit by this developer related to this file. This kind of expertise is also known as implementation expertise.

Implementation expertise. Within a codebase, developers accumulate expertise by *changing* methods when they learn the implementation details of the method. This kind of expertise has been used for past expert recommendation systems (see Section 2).

In this paper, we will introduce a different kind of expertise:

Usage expertise. Developers also accumulate expertise by *calling (using)* methods. While they might not be aware of implementation details, they know when a method can be used and how it should be called.

Both kinds of expertise can be mined from version archives. In our data collection, we scan commits for changed methods (implementation expertise), as well as for inserted methods calls (usage expertise). This data is then aggregated in an *expertise profile* for each developer (Section 3). We also present potential applications for expertise profiles, which range from recommending experts for methods with no or little history, identifying developers with similar expertise, and measuring the usage of API methods (Section 4). We conclude the paper with a discussion of future work (Section 5).

2. RELATED WORK

Our approach for expert recommendation is not the first to mine software repositories. However, all previous approaches were based on locations such as files and packages and thus, expertise was specific to a given project. By mining usage expertise instead, we get project-independent expertise (e.g., for external libraries), which is transferable *across projects.* This allows to integrate newcomers to software projects in expert recommendation. In addition our approach allows to *recommend experts for code with no or little history.*

Minto and Murphy present the Emergent Expertise Locator (EEL) that proposes experts to a developer based on the recently edited or selected files [7]. Their approach uses the coordination requirements framework introduced by Cataldo et al. [2]. 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.

Anvick and Murphy propose and compare three different ways to determine implementation expertise from bug reports [1]. Two approaches involve analyzing source repository check-in logs. Here

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR'08, May 10-11, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-024-1/08/05 ...$5.00.

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.

Mockus and Herbsleb present the Expertise Browser [8] that uses experience atoms (EA) as a measure of expertise. These EAs are gathered mining the source repository using author and change information. Then each EA can be associated with several domains (such as author, organization, technology used, 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 [6] 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.

3. METHODS AS UNIT OF EXPERTISE

Expertise is a combination of domain knowledge, skills, and characteristics that give a person the ability to solve problems rapidly and effectively in a given problem domain. In some domains there is an objective measure for expertise, e.g., sportsmen can measure time, heart rates, and lactate values to gain an objective measure of their fitness. In software development, however, there exists no objective measure to predict how well a developer will solve a given task.

In order to provide a measure for expertise in software development, our approach measures expertise on the level of methods. In an object-oriented system, methods are an object's way of providing and describing the services it offers. By changing a method, developers express that they understood the effects and the functionality of this method (implementation expertise). By calling a method, developers express that they are aware how to use the method (usage expertise).

3.1 Data Collection

Our approach can be applied to any version control system. However, we based our implementation on CVS since many open-source projects currently use it. First, we reconstruct CVS commits with a *sliding time window* approach [12]. A reconstructed commit is a set of revisions R, where each revision $r \in R$ is the result of a single check-in.

Additionally, we compute method calls that have been inserted within a commit operation R. Every commit R also has set of changed locations $C(R)$—in our case locations are method bodies but could be classes or files as well. For every location l that was changed in R, we compute the set of added method calls $M(l)$ by comparing the abstract syntax tree of l before and after commit R. As a result we obtain the set of *new calls* for a commit R

$$T(R) = \{m \mid l \in C(R), m \in M(l)\}$$

In the example below, the commit R_{ex} added calls to three methods.

$$T(R_{ex}) = \{\texttt{addTest(1)}, \texttt{worked(1)}, \texttt{refreshStatus(0)}\}$$

Table 1: Expertise profile for Erich Gamma (egamma).

Ten most frequently changed		Ten 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
`testReran`	117	`asyncExec`	24
`testRunEnded`	114	`handleFieldChanged`	24
`showFailure`	113	`postRefresh`	24
`endTest`	109	`findType`	21

The set of changed locations $C(R)$, the set of new calls $T(R)$, and the author of commits $author(R)$ serve as main input for the construction of expertise profiles, which is described in Section 3.2.

Unlike Williams and Hollingsworth [10], 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 result in high computational costs. Instead, we analyze only the differences between single revisions. As a result, our preprocessing is cheap, as well as platform- and compiler-independent; the drawback is that types cannot be resolved because only one file is investigated. As a consequence, we cannot resolve signature for called methods. Instead we identify methods calls with their names (e.g., `addTest`) and number of arguments (e.g., `(1)`). For more details on our preprocessing, we refer to the APFEL plug-in [11].

3.2 Expertise Profiles

For each developer the data mined from the version archive is aggregated in an expertise profile. Such a profile contains all methods a developer changed (implementation expertise) and called (usage expertise) as well as frequency counts. More formally, the expertise profile P for a developer d is represented by the tuple

$$(I_d, U_d, changes_d, uses_d)$$

with

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

$$U_d = \{m \mid m \in T(R), R \in \mathcal{R}, author(R) = d\}$$

and the frequency counts $changes_d : \mathcal{L} \to \mathbf{N}$ and $uses_d : \mathcal{L} \to \mathbf{N}$ defined as

$$changes_d(m) = |\{R \mid R \in \mathcal{R}, m \in C(R), author(R) = d\}|$$

$$uses_d(m) = |\{R \mid R \in \mathcal{R}, m \in T(R), author(R) = d\}|$$

In addition, we relate counts to the total number of method changed/called by a developer ($rel_changes_d : \mathcal{L} \to \mathbf{R}$ and $rel_uses_d : \mathcal{L} \to \mathbf{R}$).

Table 1 shows the expertise profile of Erich Gamma *(egamma),* which we mined from the ECLIPSE CVS archive. We ignored getter and setter methods, as well as calls to Java API. 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.

The profile in Table 1 is computed for the entire lifetime of ECLIPSE. However, it is also possible to compute expertise profiles on a weekly (or monthly) basis to show what developers have been recently working on.

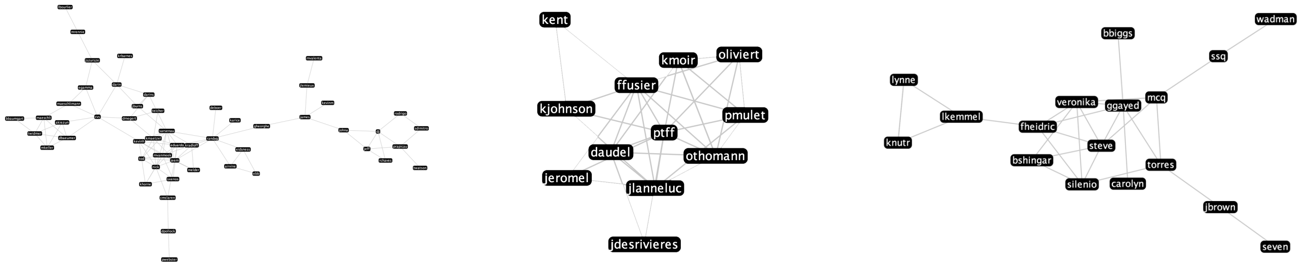

Figure 1: The three largest neighborhood components in ECLIPSE. While developers in the left component do not share a unique connecting theme, the developers in the middle component mostly use the JDT compiler and the developers in the right component the ECLIPSE user interface API.

4. POTENTIAL APPLICATIONS

Expertise profiles help to locate and recommend experts (Section 4.1), to support collaboration and communication between developers (Section 4.2), and to analyze and inform about the developers usage of APIs (Section 4.3).

4.1 Identify experts

Developers and managers can query the profiles to identify who matches a given expertise Q the best. This helps with assigning tasks, addressing questions to the right person, and organizing code review ("Who has experience with SQL?").

For example, we want to know which ECLIPSE developers have experience in *using* a given module, say the SQL part of the JAVA SDK. First, we define Q as the 358 methods that are contained in the `java.sql` package. (We manually filtered methods not specific to SQL functionality, such as `toString`.) Next we compute for every developer d the overlap between U_d and Q and rank accordingly. Here's what we get for ECLIPSE (top seven developers):

Developer d	$\lvert U_d \cap Q \rvert$
aweinand	18
carolyn	11
dj	10
bbokowski	9
othomann	8
dorme	7
dbaeumer	7

In this example, considering usage expertise has the following advantages compared to implementation expertise:

- The developer Andre Weinand *(aweinand)* never changed parts of the `java.sql` package. While he might not be experienced enough to change it, he knows enough to use it, which is what is needed in most projects.

- While implementation expertise is typically project specific, usage expertise is to some extent project independent (the parts that are referring to external libraries).

In contrast to other expertise location techniques, our approach also works for code with no or little history and in situations where for a piece of code none of the previous developers is available to change it. Using the method calls in that piece of code as our query Q, we can identify developers who should have enough expertise to understand and change this code.

4.2 Neighborhood of developers

Based on the usage profiles, we can define for every developer a neighborhood that consists of the developers with the most similar expertise. More formally, with $X = U_c \cup U_d$ we define similarity between two developers c and d as:

$$\textit{similarity}(c,d) = 1 - \frac{1}{|X|} \sum_{m \in X} |\textit{rel_uses}_c(m) - \textit{rel_uses}_d(m)|$$

The similarity is between zero and one, where zero indicates disjoint expertise profiles and one indicates identical profiles. There are many other possible ways to define similarity and in future work we will evaluate which one works best.

Using the above similarity measure, the five closest neighbors for the ECLIPSE developer *egamma* are the following:

Developer c	Developer d	*similarity*(c,d)
egamma	maeschlimann	0.1198
egamma	jszursze	0.0939
egamma	cvs	0.0815
egamma	mkeller	0.0740
egamma	tmaeder	0.0701

The highest observed similarity in our experiments was 0.2445 (between the developers *sarsenau* and *jeem*). One reason for the low similarity values is that we included all methods, including `private` ones exclusively changed and called by one developer.

Ideally, neighborhood relations can encourage communication and collaboration between developers. By considering usage expertise rather than implementation expertise, *developers from different projects can become neighbors* when they work with the same API. They can collaborate and discuss the usage of the API and maybe even team up with their projects. Neighborhood information is also a very good starting point for newcomers to a software project and helps them to direct their questions to the right person.

Furthermore, neighborhoods help to analyze the team structure, e.g., to find developers that work on related parts of the code. This information can be used to schedule meetings and create interest groups ("Which developers should be interested about SWT in ECLIPSE?") or to choose developers for training of their skills ("Which developers would benefit most from a training in this new database technology?").

Figure 1 shows a social network of ECLIPSE in which two developers c and d are connected when *similarity*$(c,d) \geq 0.08$. For brevity, we show only the three largest connected components. For one of the components, we could not identify a connecting theme, but the other ones were connected because they used the JDT compiler and the ECLIPSE user interface respectively.

4.3 Impact of API methods

In software development, there is a hidden dependency between developers who create methods (API producer) and the developer who call it (API consumer). Our approach of mining usage expertise makes these dependencies explicit and supports communication between API producers and consumers.

For API producers, we can provide information how their API is used and *by whom*. They can ask consumers for feedback on the API or inform them about upcoming changes (or serious bugs in the implementation). Moreover, they obtain information about the popularity of methods (which ones are used by many developers). This helps to prioritize and plan development efforts (e.g., refactoring): heavily used methods deserve more attention, while rarely used methods are candidates for deprecation.

A similar approach for measuring the API impact was proposed by Holmes and Walker [4], which also distinguishes between API producers and API consumers. In their work they investigate the *consumption* of the ECLIPSE API. They consider program relations like inheritance, interface implementation, and method calls to (a) inform API producers how their API is used by other developers and (b) inform API consumers how other consumers use the API. In contrast to our approach, they do not consider author information since their analysis is focused solely on source code. In particular, their approach is mostly for information purposes, while our approach can establish a communication channel between API producers and consumers—again across projects.

5. CONCLUSIONS AND CONSEQUENCES

In this paper, we proposed a new measure for expertise that is based on how developers use methods (usage expertise), which complements the expertise measure based on what methods developers change (implementation expertise). We also sketched possible applications for these expertise measures, which we will thoroughly evaluate in our future work. In addition, we plan to work on the following:

Improve precision of data collection. In our prototype, method calls are not fully resolved because we are only analyzing program fragments. We use a conservative strategy for ambiguities like overwritten and overloaded methods. We plan to use *fragment class analysis* [9] to collect more precise data.

Combining implementation and usage expertise. A combined usage and implementation expertise metric may be more accurate in assigning tasks to developers. In addition, taking both implementation and usage expertise into account, assigns different roles to developers. They can be considered as consumers and producers of methods, which can help to facilitate communication between them.

Combining expertise with tasks and bug reports. We plan to integrate expertise information with additional data sources such as tasks and bug reports. 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.

Relation between expertise and quality of code. We plan to conduct an empirical study about the effects that expertise has on the development process. For example, what are the implications of expertise on the quality of the code: "Are changes by more experienced developers less error prone?".

To learn more about our work in mining expertise, visit

`http://www.kode1061.com/`

6. ACKNOWLEDGMENTS

This research was supported by an IBM Jazz Faculty Award and by a start-up grant from the University of Calgary. Many thanks to Rahul Premraj, Christian Lindig, and the anonymous MSR reviewers who gave valuable feedback on earlier revisions of this paper.

7. REFERENCES

[1] 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*. IEEE Computer Society, 2007.

[2] 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 ACM Conference on Computer Supported Cooperative Work*, pages 353–362. ACM, 2006.

[3] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: Distance and speed. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 81–90. IEEE Computer Society, 2001.

[4] R. Holmes and R. J. Walker. Informing Eclipse API production and consumption. In *Eclipse '07: Proceedings of the 2007 OOPSLA Workshop on Wclipse Technology Exchange*, pages 70–74. ACM, 2007.

[5] 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. ACM, 1998.

[6] D. W. McDonald and M. S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *CSCW '00: Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, pages 231–240. ACM, 2000.

[7] S. Minto and G. C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007.

[8] 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. ACM, 2002.

[9] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[10] 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.

[11] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *Eclipse '06: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange*, pages 16–20. ACM, 2006.

[12] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR '04: Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, 2004.