Recommending Emergent Teams

Shawn Minto and Gail C. Murphy
University of British Columbia
Vancouver, B.C., CANADA
{sminto, murphy}@cs.ubc.ca

Abstract

To build successful complex software systems, developers must collaborate with each other to solve issues. To facilitate this collaboration, specialized tools, such as chat and screen sharing, are being integrated into development environments. Currently, these tools require a developer to maintain a list of other developers with whom they may wish to communicate and to determine who within this list has expertise for a specific situation. For large, dynamic projects, like several successful open-source projects, these requirements place an unreasonable burden on the developer. In this paper, we show how the structure of a team emerges from how developers change software artifacts. We introduce the Emergent Expertise Locator (EEL) that uses emergent team information to propose experts to a developer within their development environment as the developer works. We found that EEL produces, on average, results with higher precision and higher recall than an existing heuristic for expertise recommendation.

1 Introduction

Software developers must collaborate with each other at all stages of the software life-cycle to build successful complex software systems. To enable this collaboration, integrated development environments (IDEs) are including an increasing number of tools to support collaboration, such as chat support (e.g., ECF[1] and the Team Work Facilitation in IntelliJ [2]) and screen sharing.

All of these tools have two limitations that make them harder to use than necessary. First, the tools require the user to spend time and effort explaining the mechanism to all members of a team with whom he may want to communicate over time (i.e., a buddy list). Given that the composition of software teams is increasingly dynamic for many organizations due to agile development processes, distributed software development and other similar trends, it may not be straightforward for a developer to keep a description of colleagues on the many teams in which she may work up-to-date.[3] Second, the tools require the user to determine with whom he should collaborate in a particular situation. This requirement forces the user to have some knowledge of who has expertise on particular parts of the system.

In this paper, we describe an approach and tool, called Emergent Expertise Locator (EEL), that overcomes these limitations for developers working on code. The intuition is that a useful definition of a team, from the point of view of aiding collaboration, are those colleagues who can provide useful help in solving a particular problem. We approximate the nature of a problem by the file(s) on which a developer is working. Based on the history of how files have changed in the past together and who has participated in the changes, we can recommend members of an emergent team for the current problem of interest. Our approach uses the framework from Cataldo et. al.[2], adapting their matrix-based computation to support on-line recommendations using different information, specifically files rather than task communication evidence. EEL produces a ranked list of the likely emergent team members with whom to communicate given a set of files currently of interest.

To determine the accuracy of EEL in predicting emergent teams, we applied the approach to historical data for the Eclipse project, Firefox and Bugzilla. We found that EEL produces, on average, results with higher precision and higher recall than an existing heuristic.

We begin by comparing our approach with existing work on locating experts (Section 2). Next, we describe our approach and implementation (Section 3) before presenting our validation of the approach (Section 4). Before summarizing, we discuss outstanding issues with our approach and validation (Section 5).

[1] Eclipse Communications Framework, http://www.eclipse.org/ecf/, verified 1/8/07.

[2] IntelliJ is a Java IDE, http://www.jetbrains.com/idea/, verified 1/8/07.

[3] As one example, the Eclipse development process uses dynamic teams as described by Gamma and Wiegand in an EclipseCon 2005 presentation, http://eclipsecon.org/2005/presentations/econ2005-eclipse-way.pdf, verified 1/8/07.

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

2 Related Work

Three types of approaches have been used to recommend experts for a software development project: heuristic-based (e.g., [10]), social network-based (e.g., [11]) and machine learning-based (e.g., [1]).

Heuristic-based recommenders apply heuristics against data collected from and about the development to determine who is an expert in various areas of the system. Some approaches require users to maintain profiles that describe their area of expertise (i.e., Hewlett-Packard's CONNEX[4]) or organizational position (i.e., [7]). Unfortunately, it is difficult to keep such profiles up-to-date. During a field study of expertise location, it was found that a seven-year old profile-based system was available but the profiles had never been updated [6]. To avoid this problem, EEL does not use any profile-based information.

Other heuristic-based expertise recommenders are based solely on data extracted from the archives of the software development. The Expertise Browser (ExB), for example, uses experience atoms (EA), basic units of experience, as the basis for recommending experts [10]. Experience atoms are created by mining the version control system for the author of each file revision and the changes made to the file. A mined experience atom is then associated with multiple domains (e.g., the file containing a modification, the technology used, the purpose of the change and/or the release of the software). A simple counting of experience atoms for each domain in question is then used to determine the experience in that area. Similar to our approach, ExB equates experience to expertise. In contrast, our approach accounts for the relationships between file modifications, which we believe contain rich information about the expertise of the developer who made the change.

As another example, the Expertise Recommender (ER) by McDonald [7] was deployed using two heuristics: tech support and change history. The change history heuristic, which is related to our work, uses the "Line 10" rule that states that the revision authors are the experts for a file. These experts are ranked according to revision time so that the last developer to modify the file has the highest rank [7]. If multiple modules are selected as the target for an expertise request in ER, an intersection of the experts is performed, raising the possibility of ER producing an empty set of experts. In contrast, EEL uses the frequency of file modifications that occur together and can always produce a recommendation.

Finally, Girba et. al. used a heuristic that equates expertise to the number of lines of code that each developer has modified [3]. This approach assumes that a developer that modified a line of code is the expert for that line. The expert for a file is then the developer with the highest number of lines of code. This method could be beneficial to EEL, but would additionally require the tracking of code within a file.

A social network describes relationships between developers built using data mined from the system development (e.g., [4]). These networks often become large. As a result, many tools support queries to prune the network to show the most relevant portion; for instance, enabling the production of a view with experts in a particular area such as NetExpert [11]. This social network approach adds complexity for the user since they must be able to interpret and search the network to extract the information that they want. In contrast, the query needed to determine the experts in EEL is formed behind the scenes automatically based on what the artifacts and tasks on which the developer is working.

Machine learning-based approaches in the area of expertise recommendation have focused on using text categorization techniques to characterize bugs [1] and documents [12]. Similar to machine learning-based expertise recommenders, EEL relies on past information to form recommendations. In contrast to these approaches, EEL uses a simple frequency-based weighting to form recommendations and does not produce any general model of the activity between developers.

3 Approach and Implementation

3.1 Approach

The goal of the Emergent Expertise Locator (EEL) is to make it easier for a developer to determine with whom to communicate during a programming task. EEL displays a ranked list of other developers with expertise on the set of files that the user of EEL has recently edited or selected, their current change set. To use EEL, a developer accesses a menu on a source file that displays a ranked list of developers along with ways to initiate a communication as in Figure 1. These communication methods may be synchronous (i.e., chat) or asynchronous (i.e., e-mail). This approach aims to minimize the impact of the communication on a developer's work flow and aims to provide assistance in context; for example, a developer need not switch to an external application to perform the communication and context about the developers current state may be automatically transmitted to the expert with which communication is begun.

3.1.1 Mechanics

Our approach is based on the mechanism of using matrices to compute coordination requirements introduced by Cataldo et al. [2]. Our application of their framework involves two matrices, the file dependency matrix and the file authorship matrix, and produces a third, the expertise matrix.

1. *File Dependency Matrix* A cell *ij* (or *ji*) in this matrix represents the number of times that the file *i* and the

[4] http://www.carrozza.com/atwork/connex/about.html, verified 1/8/07

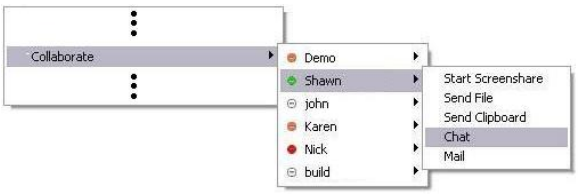

Figure 1. Context menu list of developers showing the multiple methods of communication available.

file *j* have been modified together. Since this produces a triangular symmetric matrix, EEL only records data in the upper half of the triangle to save space.

2. *File Authorship Matrix* A cell *ij* in this matrix represents the number of times an developer *i* has modified a file *j*.

3. *Expertise Matrix* This matrix represents the current experts based on the file dependency matrix and the file authorship matrix. A cell *ij* (or *ji*) in this matrix specifies the amount of expertise that developer *i* has to *j*. We consider that the higher the number in cell *ij*, the more of an expert developer *j* is to *i*. This matrix is computed using the equation:

$$\boxed{C = (F_A F_D) F_A^T} \quad (1)$$

where C is the expertise matrix, F_A is the file authorship matrix and F_D is the file dependency matrix.

When a developer selects or edits a file, EEL accesses the version control system and mines the related files and authors to populate the matrices. When the user right clicks on a file and attempts to collaborate with another developer, EEL calculates the coordination matrix on the fly to ensure up-to-date information. Since the calculation of the coordination matrix can be time intensive and since we are interested in experts only for the current developer, we modify the expertise matrix calculation to be

$$\boxed{v = (R_{F_A} F_D) F_A^T} \quad (2)$$

where v is a vector that represents the experts related to just the current developer, R_{F_A} is the row that corresponds to the current developer in the file authorship matrix, F_D is the file dependency matrix and F_A is the file authorship matrix. By using only the row that corresponds to the current developer, the matrix multiplications are reduced to simple vector calculations. Even though we are only interested in the experts for a single developer, the entire file dependency and file authorship matrices must be populated since they are required for the expertise matrix calculation.

3.2 Implementation

EEL is implemented as a Java plug-in for Eclipse. In its implementation, EEL uses matrices that are 1000 elements (files) square. This choice means that we can track up to 1000 files, enabling a substantial portion of a developer's work to be used for the recommendation of experts. Even though the matrices are fairly large, they can fill up quickly due to the number of related files per revision of a file. To mitigate this problem, EEL uses a least recently used approach to determine which entries to remove from the matrix once it becomes full, allowing the files that are either related or viewed more often to remain in the matrix longer. To ensure that the files that a developer has worked on (the current change set) remains in the matrix, they are removed only if they occupy 50% of the matrix. The current change set, the files which the developer has selected or edited, is treated differently since the set contains information that directly pertains to the developer's current work.

Since the files in the change sets provide the basis for the determination of expertise within EEL, it is necessary that they provide accurate information. Ying and colleagues noted that while mining software repositories, change sets containing over 100 files are often not meaningful since they usually correspond to automated modifications, such as formatting the code or changing the licencing [13]. After inspecting the change logs for several projects, most notably Eclipse and Gnome Evolution, we noted that this is true of most change sets with over 50 files. With this knowledge, we were able to limit EEL from mining information from change sets with over 50 files in it. This choice ensures that irrelevant related file data does not pollute the file authorship and dependency matrices.

The time required for EEL to produce a recommendation is dependant on two main factors. The first factor is the speed of the repository from which EEL accesses the author and related file information. This speed is affected by many factors such as network speed, repository size and server load. Generally, these systems can provide the information that is required by EEL quickly, therefore, it is not a major factor in the usability of EEL. The second factor is the speed of the calculation of the expertise matrix. Since the expertise matrix is computed when the user opens the menu, it is the main factor in producing a recommendation quickly. On a 2.13Ghz Core 2 Duo system with 2Gb of memory, the calculation of the expertise matrix with 1000 files and 1000 developers takes 891ms using the vector calculation approach.

4 Validation

Ideally, we would validate EEL by gathering statistics about the accuracy of EEL's recommendations as developers use the tool as a part of their daily work. Such an evaluation requires a moderately-sized, preferably distributed,

development team. Engaging such a team in an evaluation is difficult without any proven information about the effectiveness of the technique. To provide initial evaluation information, we have thus chosen to apply the approach to the history of existing open-source systems. We use information about the revisions to files stored in the version control system of a project to drive our approach. We use the communication patterns recorded on bug reports as a partial glimpse into the collaborations that actually occurred between the developers. Because we have only a glimpse into the communication that occurred during the project, the results we provide in this section are essentially a lower-bound on the accuracy of the recommendations provided.

4.1 Methodology

Our validation method involved selecting a bug of interest and recreating the development state at that time by considering only source code revisions that were committed before the bug was closed. We used a determination of the files required to fix the bug to populate the matrices and determine the recommendations. We then compared our list of experts to those who had communicated on the bug report, as determined through comments posted to the bug report. Since the communication recorded on a bug report largely discusses the issue underlying the report, the developers involved in this discussion either have expertise in the area or gain expertise through the discussion.[5]

To perform this validation, we needed to determine a set of bugs with a sufficient number of recorded comments to infer communication amongst developers and with associated revisions of the source files that "solved" the bug, the resolving change set for the bug. We searched through all of the bugs marked as resolved and fixed for reports with ten or more comments and where at least five different developers had recorded comments. For the validation, we retained only the comments provided by contributors, discarding the others as they are not relevant to providing a lower-bound on the contributor communication. We used a standard approach (see Section 4.2) to determine the resolving change set for a bug and ensured that all change sets considered between three and nine files. We chose a range of change set sizes to enable evaluation across a range of situations.

EEL is intended to be used as development proceeds. To mimic development in this validation, we used the following process:

- *Create three subsets of the resolving change set*

  Given the resolving change set for the solved bug, we create three change set sized subsets ($\frac{1}{3}$ of the change set, $\frac{2}{3}$ of the files, and the entire change set) to test how well EEL performs in finding experts given less information than what is needed to fix the given bug. We choose the files for each subset randomly with the constraint that at least one file in each subset must not be an initial revision when the bug was fixed to ensure that we have some history from which EEL can recommend emergent team members. Random subset formation is necessary since we do not know in what order a developer may have modified the files used to solve the bug.

- *Partition the comments in the bug into three groups*

  We partition the comments in the bug into three approximately equal groups based on the date of the comment. The first group has the oldest comments while the last group contains the newest ones, enabling us to mimic how development occurs.

  When no developers communicate in a comment partition, the precision would always be 0% and the recall would be incomputable since it would be divided by 0 (see below). To rectify this situation, we chose to discard these bugs from our final dataset.

- *Apply EEL to each combination of comment partitions and change set subsets*

  We apply EEL to each of the nine cases resulting from combining the comment partitions with the file revision subsets (see Figure 2) and evaluate the precision and recall of the recommendations produced by EEL. Specifically, for each case we,

  1. find the last revision of each file (in the change set subset) before the earliest comment in the comment partition,
  2. apply EEL to the file revisions in the change set subset obtained in the previous step to produce an ordered list of emergent team members,
  3. determine who all of the commenters were in the bug partition, forming the set of relevant developers,
  4. compute the precision, representing the percentage of correctly identified team members (3), and recall, representing the percentage of potential team members correctly identified (4).

$$\boxed{Precision = \frac{\#\ Appropriate\ Recomendations}{Total\ \#\ Recomendations}} \quad (3)$$

$$\boxed{Recall = \frac{\#\ Appropriate\ Recomendations}{\#\ Possibly\ Relevant\ Developers}} \quad (4)$$

The $\#\ Appropriate\ Recomendations$ is the number of developers recommended by EEL

[5] Communication on a bug report unrelated to the underlying issue is typically moved into another bug report.

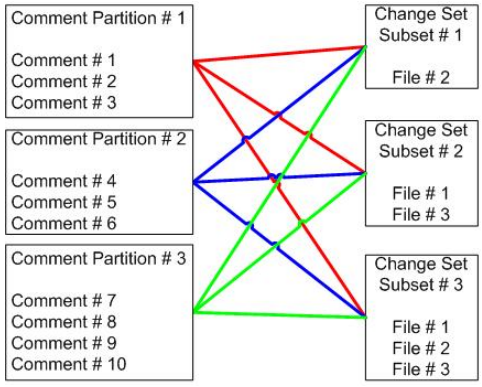

Figure 2. The 9 cases for validation.

that commented on a bug report, the $Total\ \#\ Recomendations$ is the number of recommendations that EEL made and the $\#\ Possibly\ Relevant\ Developers$ is the number of developers that communicated on the bug report.

In our validation, we also compare EEL against the "Line 10" rule that was used in the Expertise Recommender (ER) [7]. The "Line 10" rule is when the last person that modified the file is considered as the expert in that file. ER extends this approach to rank all of the developers that have modified the file by their last edit date. If multiple files are selected, the expert lists are computed separately for each file and then an intersection of the lists is performed to produce the final expert list.

After a preliminary run of the validation on EEL, it was noticed that some of the precision and recall values were 0%. Further investigation revealed that the files that changed to fix a bug may have not been created prior to the dates of earlier comments. This means that EEL was unable to get any historical data for the file since it did not exist and therefore EEL was unable to produce a list of experts. This situation can occur even if the file is not new as of that change set since development is ongoing and the file that needed to be changed was added after the bug was commented on, but before it was fixed. For these cases, we report an optimistic and pessimistic case precision and recall. The optimistic case precision and recall is 100% and it is appropriate since if the file did not exist at the time, producing no experts is correct. On the other hand, since EEL was unable to produce any experts, we can consider the pessimistic case and assume a precision and recall of 0%. These optimistic and pessimistic case values are only used if EEL is unable to produce a list of experts. If EEL is able to produce a list of experts, only one precision and recall value is given and they are computed using equations 3 and 4 as described earlier.

Another thing that we noticed upon the preliminary run of the validation was that the performance of the recommendations of EEL was lower than the "Line 10" rule for some projects. Investigation into this problem revealed that many of the developers contributing to these projects committed only a few changes during a small time period then never committed again. Since EEL considers the entire project history, the addition of authors that are no longer active affects EEL's recommendations. The "Line 10" rule is not impacted by this case since it ranks experts by their last authorship date, ensuring that the most recent authors are recommended first. In contrast, EEL uses the frequency of file modifications to produce recommendations. To ensure EEL provides relevant recommendations, we apply it to the last twelve months of development history. To be fair in our comparison, we use the same underlying data for the "Line 10" rule. The use of a limited amount of recent data from the project archives has limited effects on the results of applying the "Line 10" rule because this approach already ranks the most recent activity higher.

4.2 Data

We used three existing open-source software projects, the Eclipse project[6], Firefox[7] and Bugzilla[8] in our validation. Eclipse is an open-source platform for integrating tools implemented in Java, Firefox is a popular open-source web browser and Bugzilla is an open-source issue tracking system. These projects were chosen because they each have a number of developers who have been active in committing code and bugs in their histories and there is a sufficient amount of data to run the validation after we apply the constraints we outlined in the previous section.

To form appropriate change sets for these projects, we imported each project into Subversion using cvs2svn. Cvs2svn is a python script developed along with Subversion (SVN) by the Tigris.org[9] community. Cvs2svn has many passes that it uses to prepare the SVN repository and to determine the change sets associated with each of the CVS files, ensuring that the imported data is robust and correct. Cvs2svn creates change sets using an algorithm that inspects all files for ones that are checked in by the same author with the same check-in comment and are close in time, similar to that used by Mockus et. al. [9]. In contrast, cvs2svn uses a five minute window of time unlike the three minutes used by Mockus et. al.

In selecting bugs for the validation, we used those marked as closed and fixed within the past two years. Our selection criteria ensures that the fix is fairly recent, and that it was actually committed to the repository. If the status of a bug was not closed and fixed, it could still be under development, be a duplicate of another bug that has an unknown

[6] http://www.eclipse.org, verified 1/8/07.
[7] http://www.mozilla.com/en-US/firefox/, verified 1/8/07.
[8] http://www.bugzilla.org/, verified 1/8/07
[9] http://www.tigris.org, verified 1/8/07.

status, or it may be marked not a bug.

Since most open-source projects associate a bug with a single change set, many of them require that the identifying number of the bug that was fixed be entered into the comment of a commit to the version control system. Using this knowledge, we were able to map a bug to a single commit so that we could recreate the development state needed to fix the bug in question. To perform this mapping, we searched each of the log messages obtained from Subversion (one per revision) for a reference to one of the bugs that could be usable for validation. This was done by creating a log of all of the change sets stored in the version control system along with the files that were changed and searching for a string that indicates that it corresponds to a bug fix, for example, "bug 321", "fix 321" or just "321". We then matched logs with a reference to a bug to the bugs that we have determined are appropriate for validation. Any bug that did not have a reference in a log message was discarded from the validation.

After the data was filtered for bugs with at least ten comments, five different developers commenting and a reference in a log message, 2% (182) of all of the bugs fixed in the last two years for Eclipse remained, 6% (283) for Firefox and 10% (216) for Bugzilla. On average, there were two developers commenting on all of the fixed bugs with a maximum of 20. Also, 56% of the bugs are referred to in a log message associated with a revision.

4.3 Results

Since EEL can produce a varying number of recommendations, we computed the precision and recall for three different sized lists of potential team members, namely three, five and seven recommendations. These lists were obtained by taking the top parts of the ordered list produced by EEL and the "Line 10" rule. We varied the recommendations to investigate the impact of the size of the recommendations on the performance of EEL.

Figures 3 and 4 present the optimistic precision and recall values for Eclipse. Only the detailed results for Eclipse are presented to conserve space, for more information see [8]. The presented results represent a more interesting subset of the data that was collected. This subset presents the results for all of the time frames provided by the bug partitions, but only $\frac{1}{3}$ and $\frac{2}{3}$ of the files in the change set. Furthermore, the figures present only the results when five developers were recommended. We chose to focus our presentation of results on these cases, as these cases represent a developer looking for expertise prior to a problem being fixed and because the recommendation list is of a reasonable size for a developer to consider.

The results are presented in box-and-whisker plots. These plots assist in viewing the distribution of the results. The shaded box in the plot represents the second and third quartiles of the data set, whereas the lines extending above and below them, the whiskers, represent the fourth and first quartiles respectively. The large black dot represents the average value of the data and the line represents the median. Any small unshaded circles that are located above or below the whiskers represent outliers that are well outside the range of common values. On the results graphs, the y-axis represents the percentage value of the precision or the recall. The x-axis separates each of the cases that we are interested in. A label on the x-axis that reads T1,1/3 means that it represents the first bug partition (T1) and $\frac{1}{3}$ of the files in the subset (1/3).

Table 1 shows the overall average optimistic precision and for both EEL and the "Line 10" rule for all three of the projects used in the validation.

Table 1. Avg. optimistic precision and recall.

	Precision		Recall	
Project	EEL	Line 10	EEL	Line 10
Eclipse	37%	28%	49%	35%
Bugzilla	28%	23%	38%	28%
Firefox	16%	13%	21%	16%

In each of the three projects tested, EEL produces higher precision and higher recall than the "Line 10" rule. On average, in 88% of the 27 different test cases for each of the three projects, EEL produced a higher precision and recall than the "Line 10" rule showing and increase in recommendation performance. As can be seen, the results for Eclipse are better than that of Firefox and Bugzilla. We believe that this is because the Firefox and Bugzilla teams have many developers that commit a small number of changes during a short time period then never actively work on the project again. In these cases, EEL has less information per developer to make a recommendation than it has for Eclipse.

Anvik and Murphy used change sets to determine expertise sets, similar to EEL, but the precision and recall were 59% and 71% respectively for Eclipse [**?**,]

It is an open question whether these precision and recall values are sufficient to create an effective tool for recommendations. We are optimistic that an effective tool can be based on this approach because McDonald's study found that people working on the project generally agreed with the recommendations provided [5]. Knowing that the "Line 10" rule performs well when strict testing is performed, we believe that the our results show that EEL provides better expertise recommendations than the "Line 10" rule.

Furthermore, there were some cases where EEL was able to produce a list of experts when the "Line 10" rule recommended an empty list. Recommending an empty list of experts does not help developers find expertise, forcing them to modify the information that they are interested in

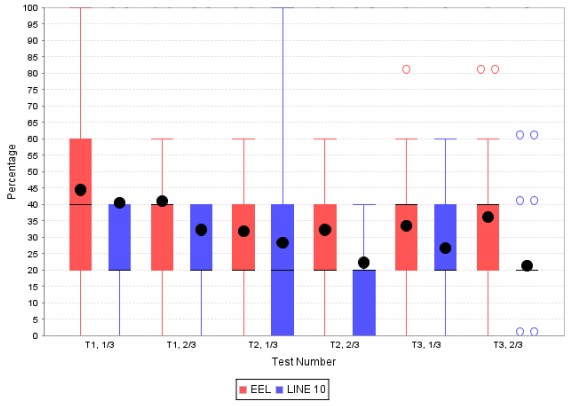

Figure 3. Eclipse optimistic precision.

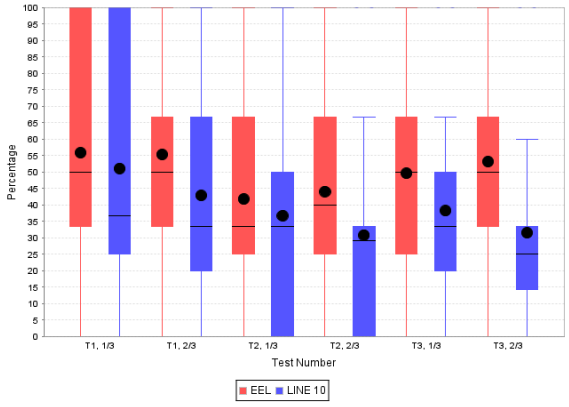

Figure 4. Eclipse optimistic recall.

until they find an expert that might be of interest. EEL always produced a recommendation if there was history in the repository for it to use. For Eclipse, this happened 6.5% of the time, Bugzilla 15.0% and Firefox 14.9%. This is a result of the "Line 10" rule performing an intersection of the authors when there are multiple files in the change set. This situation can occur when the files are relatively new, or new dependencies have been added within the files that is not reflected within the history of project.

4.4 Threats

Several factors could affect the validity of our study of EEL.

One threat is how we determined the experts to whom we compare the recommendations made by the two approaches. We used the developers who commented on the bug report as the experts for the area of the system in question. However, it is possible that the comments that were posted to the bug report were not related to the bug or were not technical in content. Either situation would mean that who we consider as experts may not actually be experts in that part of the implementation of the system. As we described in section 4.1, these situations are unlikely given how the bug reporting system is used in practice.

Our use of the bug report comment data is a lower bound on the communication that occurred during the development of the system. As a result, we may not have a complete list of the experts, or a ranking of the importance of each of the developers with respect to a bug.

Another threat is the authorship information used to recommend experts. In some cases, the authorship information we use may not indicate the expert; for instance, when a community contributor submits a patch that is applied by a committer. Since the number of such cases is typically small compared to the number of commits, we feel that this is not an issue.

The username to e-mail mappings could threaten the internal validity of the evaluation of EEL. The mapping between e-mail and username is a difficult problem and has no easy solution. Since we are comparing the performance of EEL to the performance of the "Line 10" rule, this should not affect the outcome. Both of the methods use the mappings similarly and compare to the same set of potential experts. This means that if one of the mappings is incorrect, it will affect both systems equally, therefore, not affecting the comparison between the two.

One potential threat to external validity is that we only considered open-source projects. This could be an issue since the processes used in a corporate environment might be different than that of an open-source community. As the projects that we studied involve professional developers and the systems developed are of high-quality, we believe that the processes used and the structure of the projects are similar to those in a corporate situation.

The use of mature projects could also be a threat to the generalizability of the validation. We feel that this is not a threat since any project that is new does not contain the information that is needed to provide recommendations. There is no easy way to test the validity of a recommendation tool on a new project since the project history is non-existent, therefore no information can be collected about the experts of the system. Furthermore, a recommender would produce moot results since only a few people have edited the files, meaning that they are the creators of that file and therefore the experts. On a new project, a profile-based expertise recommender would perform the best since developers can list their area of expertise.

5 Discussion

5.1 Limitations

A limitation of EEL's approach is that it is unable to easily work with many traditional version control systems like CVS and RCS. This limitation is due to these systems maintaining commit information on a per file basis; therefore, not containing any information pertaining to the files

that changed along with it. This means that we are limited to newer version control systems such as Subversion since they support atomic commits across a number of files. Tools and methods exist for extracting change sets from CVS, but it is infeasible to run this every time data is mined for a file. As an alternative, an external tool could be periodically run to extract this information, but this is an intensive operation and therefore would create the need for a server based approach which is not viable for many teams.

Another limitation to EEL is that if a new developer is added to the team, but they are already an expert, there is no support to ensure that this person is correctly recommended. If a server-based approach was used, a simple skew or replacement value could be added to augment the recommendations to ensure that this new developer is recommended. To solve this within EEL, the ability to personalize the recommendations could be added. One personalization could be the ability to substitute a expert who is recommended by EEL with a different expert specified by the user. Another use of this type of personalization would be to augment the recommendations based on the social structure of the team. A developer may prefer to talk to an expert that they know over another member that has similar knowledge. This limitation could also be solved by weighting recent developer activity higher than older information. This would mean that a developer that has worked on a file more recently could be rated higher even if they are new to the team.

5.2 Future Evaluation

The next step in evaluating EEL is to deploy the tool into an active development project. Ideally, this project would have a relatively large code base (e.g., one million lines of code), follow some agile practices and communicate primarily through electronic means. We would like the team to follow some agile practices so that EEL is able to be useful and not recommend developers that are known to be experts by members of the team. It would be beneficial if the communication was done through electronic means so that we could track and analyze the communication that was initiated through EEL. Furthermore, by engaging experts in the evaluation of EEL, we would be able to perform an analysis of the false positives that were recommended and determine if EEL recommends experts who were unknown to other developers.

6 Summary

To build successful complex software systems, software developers must collaborate with each other at all stages of the software life-cycle.

The approach that we introduce in this paper eases collaboration for dynamic, especially distributed, teams by determining the composition of the team automatically so that developers do not need to spend time configuring membership lists for the many teams to which they may belong. We achieve this goal by using the context from which the developer initiates communication combined with the project history to produce a recommendation of experts related to the area the developer is currently working. This approach essentially extracts the emergent team structure from the use of the files by developers.

Using an automated validation and historical data from three different open-source projects, we found that EEL produces higher precision and higher recall than the existing rule.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of ICSE*, pages 361–370, 2006.

[2] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proc. of CSCW*, pages 353–362, 2006.

[3] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. of IWPSE*, pages 113–122, 2005.

[4] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Proc. of AMCIS*, pages 1806–1813, 2002.

[5] D. W. Mcdonald. Evaluating expertise recommendations. In *Proc. of GROUP*, pages 214–223, 2001.

[6] D. W. Mcdonald and M. Ackerman. Just talk to me: a field study of expertise location. In *Proc. of CSCW*, pages 315–324, 1998.

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

[8] S. Minto. Using emergent team structure to focus collaboration. Master's thesis, University of British Columbia, 2007.

[9] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM TOSEM*, 11(3):1–38, 2002.

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

[11] R. Sanguesa and J. M. Pujol. Netexpert: A multiagent system for expertise location. In *Proc. of IJCAI*, pages 85–93, 2001.

[12] X. Song, B. L. Tseng, C.-Y. Lin, and M.-T. Sun. Expertisenet: Relational and evolutionary expert modeling. In *Proc. of UM*, pages 99–108, 2005.

[13] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE TSE*, 30(9):574–586, Sept. 2004.