Determining Implementation Expertise from Bug Reports

John Anvik and Gail C. Murphy
University of British Columbia
Vancouver, B.C., CANADA
{janvik, murphy}@cs.ubc.ca

Abstract

As developers work on a software product they accumulate expertise, including expertise about the code base of the software product. We call this type of expertise 'implementation expertise'. Knowing the set of developers who have implementation expertise for a software product has many important uses. This paper presents an empirical evaluation of two approaches to determining implementation expertise from the data in source and bug repositories. The expertise sets created by the approaches are compared to those provided by experts and evaluated using the measures of precision and recall. We found that both approaches are good at finding all of the appropriate developers, although they vary in how many false positives are returned.

1 Introduction

As developers work on a software product they accumulate expertise. They accumulate expertise in the particular tools that they use. They accumulate expertise in the software engineering processes and practices that they use. Finally, they accumulate expertise in the code base of the software product itself. We refer to this last form of expertise as *implementation expertise*.

Knowing the set of developers who have implementation expertise for a particular part of the software system is a useful piece of information. For instance, if a developer knows who has implementation expertise for a particular source code file then they know with whom to communicate for help and advice in solving a problem with that file. Studies have shown that, depending on the phase of the development cycle, developers spend 50% [7, 13] to 70% [5] of their time communicating with others. Knowing with whom to communicate may thus help increase a developer's effectiveness.

In software engineering, there have been two significant efforts towards providing a list of experts for a source code file: the Expertise Recommender [10] and the Expertise Browser [12]. Both of these projects use a form of the "Line 10 Rule" to determine expertise. The "Line 10 Rule" is a heuristic in which line 10 of the source repository check-in log for a particular file – the line containing the user name of the person who performed the commit – is used to determine who has expertise for that source file. The Expertise Recommender uses this heuristic to present the name of the developer with the most recent expertise for the source file. The Expertise Browser uses the heuristic to gather and rank the expertise of developers for the source file.

In each of these cases, the intent is to discover either the developer or group of developers who have expertise with a specific source code artifact. The work described in this paper looks at accomplishing a similar goal for a different project artifact: the bug report.

For many software projects the primary unit of work is the bug report.[1] Knowing which developers have implementation expertise to help resolve a particular bug report has a number of uses. Similar to previous work, the list can be used to suggest with whom to communicate about a problem, before the source code is even considered or touched. The list can be used to help determine who might fix a bug, enabling the creation and evaluation of bug report assignment recommenders [1]. Finally, the list can be used to analyze if project implementation expertise is appropriately spread out across the developers.

If there existed a strong linkage between the bug and source repositories, determining the developers with appropriate implementation experience to solve a particular bug report could proceed in a manner similar to the Expertise Browser. For example, even before implementation work proceeds on a bug of interest, similar resolved bugs to the bug of interest could be located and the "Line 10 Rule" could then be applied to the source code files linked to the similar bugs. Although there exists some tools that support such a linkage, such as Rational ClearCase, many projects use tools in which this linkage is dependent on project conventions which are not always followed [1, 15]. For exam-

[1]In this paper we refer to reports that either specify a software fault or a feature request by the colloquial term "bug report".

ple, in the Firefox[2] and Eclipse[3] projects, the convention is to forge the link by referring to the report identification number in the source repository check-in comment. For the gcc[4] project, a comment is added to the bug report that lists the files that were touched by the fix.

A more subtle problem with using this linkage technique is the assumption that the developer checking in the fix is the developer who made the fix. This assumption does not hold for all projects. An example is the Firefox project. For this project, the person who fixes the bug requests that the fix be applied to the source repository and only a subset of the project community can actually commit such a submitted fix. If care is not taken to handle these cases, they can cause the linkage technique to improperly assign expertise to project members.

An alternative to using the source repository is to use the bug repository. The report activity log, comments, and other fields in the bug report can indicate which developers have been associated with a previously resolved report. This data can then be used to determine who has implementation expertise for the report.

Previous work on mining expertise has focused on either one or the other of these repositories (e.g., [1, 3, 4, 6, 10, 12]). In all of these cases, the method introduced has been indirectly evaluated by having it as part of a recommender system and then evaluating how well the system makes recommendations (see Section 5). In this paper, we provide a direct empirical comparison of approaches based on each of these repositories to information from human experts associated with the project. This direct comparison provides a baseline by which to judge new and existing approaches.

The two approaches that we compare are the use of source repository logs and the use of carbon-copy (CC:) lists, comments, and resolver information from reports in bug networks [14]. To evaluate these two approaches we used project experts from the Eclipse Platform project to provide implementation expertise sets[5] for reports in a test set. We evaluate the effectiveness of these approaches using the standard measures of precision[6] and recall.[7] As the use of the expertise set produced by an approach determines if a high precision or a high recall is more important, we don't try to emphasis one over the other in the approaches. Instead we are interested in exploring the relative strengths and weaknesses of the two approaches. We found that both approaches had low to moderate precision, ranging from 39% to 59%, and high recall, ranging from 71% to 92%. The bug report approach was found to be a good alternative to the source repository approach with a slightly worse precision being traded off for a higher recall.

This paper makes two contributions. First, since existing approaches to mining implementation expertise do not produce perfect results, it is reasonable to expect researchers to continue to develop approaches in this area. This paper provides input into this future research by establishing the effectiveness of the basic approaches against expert information. Second, since experts can be difficult to access for the open-source projects that are often used in academic research in this area, it provides a baseline of what may be possible with existing approaches; new approaches that exceed the existing precision and recall may then be more likely to produce better results in actual use.

We begin by providing some background and describe the two approaches for mining implementation expertise that we are investigating. We then provide a description of our experiment and present our results. We finish the paper with a discussion of our findings and of related work.

2 Background and Approaches

We mined data from two different types of project repositories to determine developer expertise: the source code repository and the bug repository. In this section we provide an overview of some of the data found in these repositories and how we used it to construct implementation expertise sets.

2.1 Source Repository Check-in Logs

When a developer submits code to a source repository (also known as 'checking-in' or 'committing'), the repository logs the submission. The submission log entry includes various pieces of information, such as the file that was checked in, its revision number, the check-in date and time, and the check-in comment made by the committer. The source repository software allows a user to view all the log entries for a specific file.

Using the source repository check-in logs to create an implementation expertise set for a bug report involves three steps.

The first step is to establish the linkage between the bug report and the source repository. For projects that use tools that provide this linkage, this step is trivial. For other projects, this requires knowledge of the project's linkage convention. If the convention is to refer to the bug report identification number in the check-in comment, then establishing the linkage involves searching the source repository logs for that number.If the convention is to list the changed files in the bug report, this list is extracted from the report.

[2]`http://www.firefox.com`, verified 19/01/07

[3]`http://www.eclipe.org`, verified 19/01/07

[4]`http://gcc.gnu.org/`, verified 20/01/07

[5]We use the term 'set' to indicate that the experts and the approaches did not produce ranked lists.

[6]Precision is a measure of how many of the developer names in the set were correct.

[7]Recall is a measure of how many relevant developers are returned by the approach.

In either case the result is a list of source files that forms the *change set* for the report [16].

The next step is to determine the *containing module* for each source file in the change set. The containing module may refer to the file itself or some higher abstraction such as its package in the case of software implemented in Java. This step recognizes that it may be unrealistic to just examine source files for a particular fix when determining expertise. Developers who work with associated files, such as those in the same package, also have some level of implementation expertise that may be relevant, and this expertise should not be discounted. In this work we compare the use of two different definitions for a containing module: a Java source file and its package. This choice allows us to test the hypothesis, under one condition, that using a more general containing module will provide a better expertise set.

Next, a set of developers who had previously committed changes to the modules is compiled. This set is constructed by analyzing the revision history of each file in the module and using the "Line 10" heuristic on all the log entries for the module.

Finally, the set is filtered to remove the names of developers who are no longer relevant. Explanation of how we obtained the set of relevant developers is deferred to Section 3 where we explain the experiment.

2.2 Bug Reports and Bug Networks

A bug report contains many pieces of information in the form of pre-defined fields, free-form text, attachments, and dependencies.

To create a list of developers with implementation expertise for a bug report, we first generate the bug network that contains the report (see Section 2.2.1). Next, we extract the following information from each report in the network:

- the names of the people in the carbon-copy (CC:) list of the bug report (Section 2.2.2),
- the names of the people who added comments to the bug report (Section 2.2.3), and
- the name of the developer who fixed the bug (Section 2.2.4).

We take these lists from each report in the network and merge them to form the implementation expertise set for the bug report. As with the source repository check-in approach, the set is filtered to remove any irrelevant names.

2.2.1 Bug Report Networks

A *bug report network* is a collection of inter-related bug reports formed when members of the development community assert duplication, dependency, and reference relationships between two bug reports. A *duplication* relationship states that two bug reports describe the same problem. A *dependency* relationship states that one bug is "blocking" or "depends on" the resolution or testing of another bug. Both duplication and dependency relations are formal, symmetrical relationships, meaning that bug repositories have an established convention for specifying the dependency, such as setting a particular field, and that the relationship is bidirectional. A *reference* relationship is an informal relationship whereby a community member adds a comment to the bug report such as "See bug #2007", "My fix might also help fix bug #2007." or "Should bug #2007 be added to this bug?". Sandusky and colleagues found that 65% of the reports they examined[8] were related by one of these three relationships [14].

Figure 1 shows a bug network that contains all of these types of relationships. The figure shows a situation where Report B "blocks" the resolution of Report A (and therefore Report A "depends on" Report B), Report C describes the same problem as Report A and has been marked as a duplicate, and a comment in Report A refers to Report D.

Figure 1. Forms of bug networks.

2.2.2 Bug report CC: lists

The carbon-copy (CC:) list of a bug report is a list of individuals who want to be notified when changes are made to the report. There are a variety of reasons that someone would be interested in changes to a bug report. If the bug report represents a problem, the individuals may be encountering the problem and want to know when the problem is fixed. If the individual is a developer for the project, they may be interested because they are working on a bug for which this bug is blocking their progress. Another reason is that perhaps a more senior developer would like to keep tabs on the work of a junior developer who he is mentoring.

With respect to implementation expertise, the CC: list is a noisy data source. The list is commonly polluted with the names of people who have no project expertise, but are simply interested in the report. It is therefore necessary to clean up this data by filtering out names of individuals who are not developers for the project. In the approach we test, we defer this filtering until the lists from the three data sources are combined.

[8]Sandusky and colleagues examined a sample of 385 reports drawn from a population of more than 182,000 bug reports opened over a five year period.

2.2.3 Bug Report Comments

Discussion between developers about the best way to approach fixing a bug or the consequences of a particular fix are common [8]. In a distributed development environment these discussions are frequently reflected in the comments of a report, although other means of communication, such as email and instant messaging, are possible and are known to be used [13]. In our work, we assume that all communication about a particular bug is restricted to the bug report. We do this for two reasons. First, we are interested in generating a reasonable implementation expertise set exclusively from bug report data. Second, ensuring that all formal and informal communication about the bug report is captured is not feasible. This assumption therefore sets a lower bound on the number of people with implementation expertise for the report.

We extract the name directly from the comment section of the report as comments are labeled with the name of the person making the comment. As with CC: lists, comments represent a noisy data source and must similarly be filtered, but we defer this filtering until the sets are combined.

2.2.4 'Who Fixed' Heuristics

The most direct way to determine implementation expertise for a bug report is to know who fixed the report. There are two obvious techniques for determining who fixed a particular bug. The first is to examine the source repository check-in logs to discover who checked in the fix. However, as explained in Section 1, using this technique is problematic as it assumes that the person committing the change is the person who made the change.

The second technique is to use the value of the `status` and `assigned-to` fields in the bug report. This approach is also problematic. The problem is that projects often use the `status` and `assigned-to` fields of a bug report for additional purposes than who is assigned to fix the problem. For example, in both the Eclipse Platform and Firefox projects, the value of the `assigned-to` field does not initially refer to a specific developer. Instead new and unconfirmed reports are first assigned to a default email address before they are assigned to an actual developer.[9] For reports with a trivial resolution, such as "duplicate", the `assigned-to` field is often never changed. Another use is to pass the report on to a quality assurance team member for verification of a submitted patch. After the patch is verified, the report is marked fixed, but the `assigned-to` field does not refer to the correct developer.

As a result of these problems, we found the need in our previous work [1] to develop a set of project-specific heuristics for determining who fixed a particular bug. These heuristics can be derived either from direct knowledge of a project's process or by examining the logs of a random sample of bug reports for the project. We took the latter approach resulting in a set of heuristics for various projects. We provide two examples of heuristics for the Eclipse project here.[10]

1. If a report is resolved as FIXED, it was fixed by whoever marked the report as resolved.
2. If a report is resolved as DUPLICATE, it was resolved by whoever resolved the report of which this report is a duplicate.

3 Evaluation of the Approaches

To evaluate the two approaches, we used developers (hereafter referred to as 'experts' for clarity) to provide the correct implementation expertise sets for a test set of bug reports. The effectiveness of each approach was then found by comparing the sets generated by the approach to those from the experts. We used bug reports from the Eclipse Platform project for our test set and experienced developers from three of the project's components as the experts. Our methodology for the evaluation was:

1. Determine a set of bug reports to use as our test set (Section 3.1).
2. Have experts from the project construct expertise sets for each report (Section 3.2). Sets for the same bug report were then unioned to form the set used for comparison.
3. Apply the two approaches to the reports in the test set.
4. Compare the results from the two approaches to the sets generated by the experts (Section 3.3).

3.1 Test Set

For the test set, we selected reports from the Eclipse Platform project that met the following four criteria:

1. The report was resolved as fixed in the one month period of June 1, 2006 to June 30, 2006 inclusive. We wanted to use recent reports to enable appropriate recall by the developers. This time period also represents a very active time for the project as it was the month before the last major version (v. 3.2) release of Eclipse. During this period there were over 300 reports resolved as fixed for the project. This is important as

[9] A user name in the Bugzilla system is an email address.

[10] The full set of heuristics for Eclipse and other projects is available on-line at `http://www.cs.ubc.ca/labs/spl/projects/bugTriage/assignment/heuristics.html`.

Table 1. Number of test set reports broken down by component.

Component	# Reports	# Developers	# Responses
SWT	11	11	2
UI	10	30	2
Debug	8	6	1

the remaining criteria reduces the potential candidates significantly.

2. The report has an explicit linkage to source repository logs entries (see Section 2.1).

3. The report is part of a bug network of up to 30 reports in size. We placed an upper limit on the number of reports because as the network grows beyond a certain size it contributes less relevant information. An alternative to choosing a fixed size for the network is to limit the reports in the network based on their relational distance to the the test report.

4. The report was for an Eclipse Platform component that has five or more reports that meet the previous three criteria of resolution date, explicit source repository linkage, and network size. As development teams for the project are structured around the project's components, this criteria ensures that there are a sufficient number of test reports for analysis by experts.

Using these criteria, we obtained a set of 29 reports. The reports were grouped by component for the collection of the expertise sets (see Table 1).

3.2 Expert Generated Implementation Expertise Sets

Experts for the UI, SWT, and Debug components were provided with a small application designed for collecting the expertise sets. The expert was asked to specify for which component they had expertise. The application then presented a sequence of between eight and eleven bug reports, depending on the component. For each bug report the expert was asked to read the summary and description of the report and choose from a set of developers. Five experts participated in the study.[11]

The set of developers that the experts chose from consisted of all the developers found to have made commits to the files of the Eclipse Platform project during the period of April 2006 to June 2006. This period of time represents the three months (inclusive) before the period from which the test reports were selected.[12] Forty-two developers were found to have made commits to the projects during this period. One name was discarded as an anomaly as the developer had been very active in the repository well over a year ago in the past and then made a single commit during this time. He was therefore deemed to not be an active developer. We feel that this technique provides a reasonably accurate snapshot of all the developers actively working on the project, not just those targeted in the test set. As such, this set was used for filtering the sets generated by the approaches to remove the names of the individuals who were not relevant.

The constructed set of developers also provides a sufficient number of *distractors*. Adding items that distract from those that are likely relevant is a common technique for limiting the bias that may occur in a presented list [2, 9]. The second column of Table 1 shows the number of the developers we believe were working on each component. This was determined in a similar manner as determining the active developers for the project, but only logs for the particular component were used. Although the UI component is shown to have had 30 developers at this time, it is known that the component has a number of subcomponents so this number relative to a given report is likely inflated.

Unlike similar studies [9], we did not ask for a ranking of the developer's expertise relative to the problem. We assume that all developers that are selected by experts are equally capable of fixing the problem. We made this choice because we were only interested in determining who has implementation experience, not who were the best in the set.

Experts were asked to examine the summary and description of the report and then select from the set which developers they felt would have the expertise to fix the problem described. The source repository user names were mapped to the actual names of the developers for presentation clarity.

Experts who worked on more than one of the test set components were asked to run the application for each component on which they worked. After providing data for the set of reports, the expert was asked to provide any names that they felt were missing. No expert did so. Expertise sets created by different experts for the same report were combined to form the final set used in the evaluation. The amount that the experts agreed between themselves ranged from total agreement to complete disagreement. In most cases the experts would agree on one or two names for each set. There were only five cases out of the twenty-nine where the experts did not agree at all.

[11]The developers who provided expertise sets all had over two years of experience on their respective components.

[12]Expanding this time period was found to only add names for developers that appeared to not be significantly contributing to the project.

Table 2. The minimum, average, and maximum sizes of the sets from the experts and the three approaches.

	Min	Mean	Max
Expert Sets	1	4	30
SR-Change Set	1	3	6
SR-Package	4	8	13
Bug Network	2	5	12

Table 3. Average precision & recall for the different approaches of expert set generation.

	Precision	Recall
S.R. (Change Set)	59	71
S.R. (Package)	39	91
Bug Network	56	79

3.3 Results

Table 2 shows the sizes of the implementation expertise sets created by the experts and the approaches. The source repository approach using package granularity (SR-Package) was found to produce larger sets than the source repository approach using change set granularity (SR-Change Set). Given that a package granularity is more general than a change set granularity, this is not surprising. The bug network approach was found to produce sets of comparable size to that of SR-Change Set, and both of these approaches produced sets of similar size to those created by the experts.

Although the expertise sets created by the experts were fairly small (an average size of 4), there was one anomaly. For one of the test cases, an expert selected thirty different developers as having the necessary expertise. Upon closer examination of the test case, it was for a report where the visibility of a group of methods was to be changed. As this task does not require a high level of expertise, the expert chose a large number of project developers.

Table 3 shows the results of our experiment. It presents the average precision and recall for the two variations of the source repository check-in approach and for the bug network approach. The formulas for computing the precision and recall values are given in Formulas 1 and 2 respectively.

$$Precision = \frac{\#\ correct}{size\ of\ generated\ list} \quad (1)$$

$$Recall = \frac{\#\ correct}{size\ of\ expert\ list} \quad (2)$$

We found that the SR-Package approach was the best, on average, at finding the names of developers who had the relevant implementation expertise (i.e a high recall). The SR-Change Set approach was the best of the three at producing an expertise set where most of the names were correct (i.e. high precision) with few false positives. The approach using only information from the bug report produced similar results to that of SR-Change Set, with a higher recall compensating for a lower precision.

4 Discussion

4.1 Which is the Best Approach?

As stated before, we found that the source repository approach using the package as the 'containing module' tended to be better at finding all the relevant names. However, this was accomplished by producing sets that were larger than those from the other approaches (see Table 2) and contained many false positives. The source repository approach using change sets was the best at producing sets that were mostly correct, but the approach missed almost a quarter of the relevant developers on average.

Deciding which is the best approach depends on how the expertise set is going to be used. We present two applications where expertise sets would be used and discuss the relative merits of using the different approaches. As the approach using bug report data produces similar results as the source repository approach using change set granularity, we will only discuss the trade-offs between the two source repository approaches.

4.1.1 Recommending Experts

If the expertise set is to be used for recommending experts, then the SR-Change Set approach seems the best, as it will produce a set with the fewest false positives. However, it will miss, on average, about a quarter of the relevant developers. For the application of expert recommendation, this is acceptable as the user is looking for an expert, not all the experts. If the SR-Package approach was used, then the set of recommended experts would contain many incorrect names and the user would likely quickly get frustrated with the recommender. This result then indicates that both the Expertise Recommender and Expertise Browser system used the appropriate approach for creating their recommendations.

4.1.2 Evaluating an Expertise Recommender System

A common technique for evaluating a recommender system is to determine its average precision and recall with respect to a testing set in a similar manner to how we evaluated the two approaches. The key item of information needed to compute the precision and recall is the set of correct answers for each test case. If the recommender system under evaluation is for recommending expertise, then the set of all experts for the test case is needed. If the SR-Package approach is used to generate this set, then nearly all of the experts will be listed. This will mean that when the precision metric for the recommender system is calculated, it will be close to the value that would have been obtained had experts provided the expertise sets. However, as the approach produces sets that contain many false positives, the recall value will be lower than the true value as the denominator of the recall formula is the size of the set of experts and this value will be larger than its true value. The converse is also true. If the SR-Change Set approach is used to generate the expertise set for evaluation, then the recommender's calculated precision will be lower than the true precision of the recommender, but the recall will be closer to its true value. Choosing which approach to use for generating the expertise set depends on if an accurate precision or recall value is of most importance in the evaluation.

4.2 Using the Approaches for Other Projects

For practical reasons, we used only one project, an Eclipse project, to evaluate the approaches. Here we discuss how this project's characteristics may or may not allow these results to extend to other projects.

4.2.1 Using Source Repository Check-in Logs

As mentioned previously, one of the assumptions with the source repository check-in approach is that the individual who commits the change is the one who made the change. For the Eclipse Platform project, this assumption holds. However, for a project which uses a process in which fixes are reviewed and contributers must request that their changes be checked in by a committer, the source repository approach will be naturally misleading. Our results show that using the bug network approach is a reasonable alternative that can be used for determining implementation expertise for such a project.

4.2.2 Determining the Active Developers

As both approaches derive the expertise set from historical data, an important step for both approaches is the filtering of the generated expertise sets to remove the names of individuals who are not currently active on the project. However, this step requires knowing who are the active developers. Our approach to obtaining this information was to use source repository check-in information. For the Eclipse project, this is appropriate as developers tend to also be committers on the project. However, for a project where fixes are checked-in by quality assurance representatives as opposed to the developers, this is not appropriate, and an alternate means would be necessary.

5 Related Work

To our knowledge, there has been only one other effort towards evaluating how well an implementation expertise approach works. The Expertise Recommender [9] was evaluated in a manner similar to how we evaluated the two approaches. The heuristics used in the system were derived from the work practices of two groups. Users from the two groups were presented with scenarios and a list of six recommendations with three recommendations being from the Expert Recommender and three being distractors. Participants were asked to rank-order the recommendations, with a blank space provided to allow the participant to add a name if they felt there was one missing from the list. Points were awarded to each list based on if a recommendation appeared in the top half or a distractor appeared in the bottom half of the ranked list. The mean score for the lists was 51.82 points out of a maximum of 72 points. This was found to be significantly better than random, and so the system was deemed to work reasonably well.

Although the Expertise Browser [12] used the 'Line 10' heuristic, it did not evaluate how well the heuristic worked. Instead, the Expertise Browser was evaluated by deploying it to sites of two different companies and collecting usage information. The authors found that the tool was most used at the site where the developers were least familiar with the product and the team was small so that there were few experts. Qualitative user feedback was also collected and revealed some unexpected uses of the tool.

As the heuristics in the Expertise Recommender were evaluated using a point system, and our evaluation used precision and recall, we cannot directly compare our results to that work. Similarly, as the Expertise Browser did no heuristic evaluation, we cannot make any comparisons to that work. However, the evaluation of both of these systems do contain results that are similar to our findings. In the evaluation of the Expertise Recommender, McDonald found that participants agreed the most with the 'Line 10' heuristic. In the evaluation of the Expertise Browser, a manager commented that he learned more in a few minutes about who actually did what on a particular release than when he was actively managing the release. Both of these results are similar to our results that the source repository approach can provide a reasonable approximation of implementation expertise.

Most recently, the Emergent Expertise Locator (EEL) by Minto and Murphy [11] used the SR-Change Set approach to determine expertise for emergent development teams. Like this work, the measures of precision and recall were used for evaluating expertise predictions. They reported a precision and recall of 37% and 49% respectively for the Eclipse project. Although these results appear to conflict with ours, the data set used by Minto and Murphy covered all of the Eclipse IDE projects, whereas our work only examined data for one of the Eclipse IDE projects and used a much smaller data set.

6 Conclusion

This paper presented an empirical evaluation of two approaches for determining who has implementation expertise for a bug report using data from two types of repositories. The first approach determined implementation expertise by examining the source repository check-in logs for the 'containing modules' associated with the bug report. The second approach used data from the bug reports in the bug's network to determine expertise. The expertise sets created by the two approaches on a set of reports from the Eclipse Platform project were compared to the expertise sets produced by project experts. We found that depending on what was wanted from the expertise set, different approaches were best. If it is important to ensure that all developers with the necessary implementation expertise are found, then the source repository approach using the package as the 'containing module' was the best. If it is important that the expertise set contain the fewest false positives, then the source repository approach using change set as the 'containing module' was the most appropriate. The bug report approach was found to be a good alternative to the source repository approach using change sets, especially when the source repository data is known to not be accurate.

As experts can be difficult to access for the open-source projects that are often used in academic research in this area, this work provides a baseline of what may be possible with existing approaches; new approaches that exceed the existing precision and recall of these approaches may then be more likely to produce better results in actual use.

7 Acknowledgments

This research was funded in part by NSERC and in part by an IBM Eclipse Innovation Grant.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the 28th Int'l Conference on Software Engineering*, pages 318–370, 2006.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.

[4] D. Čubranić and G. C. Murphy. Automatic bug triage using text classification. In *Proc. of Software Engineering and Knowledge Engineering*, pages 92–97, 2004.

[5] T. Dearco and T. Lister. *Peopleware: Productive projects and teams*. Dorset House Publishing., New York, 1987.

[6] G. A. Di Lucca, M. D. Penta, and S. Gradara. An approach to classify software maintenance requests. In *Proc. of the Int'l Conference on Software Maintenance*, pages 93–102, 2002.

[7] J. D. Herbsleb, H. Klein, G. M. Olsen, H. Brunner, J. S. Olsen, and J. Harding. Object-oriented analysis and design in software project teams. *Human Computer Interaction*, 10(2&3):249–292, 1995.

[8] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of the 29th Int'l Conference on Software Engineering*, 2007. To appear.

[9] D. W. McDonald. Evaluating expertise recommendations. In *Proc. of the 2001 Int'l ACM SIGGROUP Conference on Supporting Group Work*, pages 214–223, 2001.

[10] D. W. McDonald and M. S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *Proc. of ACM Conference on Computer Supported Collaborative Work*, pages 231–240, 2000.

[11] S. Minto and G. C. Murphy. Recommending emergent teams. In *Proc. of 4th Int'l Workshop on Mining Software Repositories*, 2007. To appear.

[12] A. Mockus and J. D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proc. of the 24th Int'l Conference on Software Engineering*, pages 503–512, 2002.

[13] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, pages 38–45, July 1994.

[14] R. J. Sandusky, L. Gasser, and G. Ripoche. Bug report networks: Varieties, strategies, and impacts in a f/oss development community. *Proc. of 1st Int'l Workshop on Mining Software Repositories*, pages 80–84, 2004.

[15] C. C. Williams and J. K. Hollingsworth. Bug driven bug finders. In *Proc. of 1st Int'l Workshop on Mining Software Repositories*, pages 70–74, 2004.

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