Measuring Developers' Expertise Based on Version Control Data

Anett Fekete*, Máté Cserép* and Zoltán Porkoláb*
* ELTE Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary
{afekete, mcserep, gsd}@inf.elte.hu

***Abstract* – Developers' fluctuation in the lifetime of a software product might deteriorate the understanding of the source code to a level where developer expertise of some modules drops to a dangerously low point. It is important for the project management to identify such critical modules to avoid complete knowledge loss. This paper presents a developer-centered static analysis tool that is intended to show individual expertise in large software projects. The expertise value is computed for each file through repository mining of the version control system of the project. The calculated value is based on the quality of commits per developer. The results of the proposed method have been validated on the CodeChecker open-source project, comparing against the findings of a user questionnaire filled by developers of the project on their expertise.**

Keywords - developer expertise, version control, repository mining, code comprehension

I. INTRODUCTION

In case of long-running software projects, the fluctuation of developers is inevitable, whether we talk about smaller projects with only a few developers at once, such as a university lab project, or large, industrial projects that count tens or hundreds of developers. When fluctuation is so high, there is always a risk that certain components of the software suffer neglect. If all or even most developers that have knowledge about a component leave the project, maintenance problems might emerge, and it will cause much more problems to debug or develop the component than it would be in possession of decent expertise [1].

During development, lots of questions emerge that concern other developers, especially when we need to find out about the development reasons behind a certain part of the code. Developers might also need to know the authors of a certain code who presumably know the most about the code and are able to help with understanding and debugging the code in question. These questions might be hard to answer, especially in large companies, where dozens of programmers develop a software. Nowadays, using some type of version control system for our projects like Git or SVN is fundamental. Repositories are capable of storing every piece of information that is bound to development since the start of the project. Naturally, the information we need is not stored explicitly but in the form of commits that can be processed and analyzed.

This work studies the following research questions:

- The quantity of modifications in a commit is easy to measure as the number of modified lines. The quality of modifications is more difficult to determine since they have to be classified based on the type of modifications such as refactoring, new functions, new classes, etc. Quantifying the quality of a modification in a file depends on the programming language, thus requires static analysis of the language. This makes quantification harder in case of multilingual projects where we have to perform static analysis in every language that is used in the software. Advanced software similarity checking software has the capability of analyzing several languages and gives accurate results of comparison between different states of a file. Similarity checking is suitable for comparing the previous and new versions of source files. They provide a quality measurement through a selected comparison approach [2]. The results of similarity checking can be used to classify modifications according to their impact. **RQ1**: *How to identify insignificant commits (i.e. commits with no substantial changes, such as adding or removing comments, renaming identifiers, adding copyright information, etc.)? Are software similarity checkers capable of effectively identifying such modifications?*
- All developers have a notion of their knowledge in their current project(s). Source code is usually the atomic part of this knowledge which is extended by their comprehension of software architecture, design patterns, coupling information, etc. **RQ2**: *How do developers' contributions to the source code correlate? Do developers correctly estimate their own knowledge in the software projects they work on? If not, do they tend to over- or underestimate their knowledge?*

We developed a tool to parse and analyze the Git commit history to answer the above questions through empirical measurement and evaluation. The newly developed *competence plugin* has been implemented as part of the open source code comprehension supporting software CodeCompass [3]. The plugin calculates the quantity of modifications as lines of code (LOC), and the

PREPARED WITH THE PROFESSIONAL SUPPORT OF THE DOCTORAL STUDENT SCHOLARSHIP PROGRAM OF THE CO-OPERATIVE DOCTORAL PROGRAM OF THE MINISTRY OF INNOVATION AND TECHNOLOGY FINANCED FROM THE NATIONAL RESEARCH, DEVELOPMENT AND INNOVATION FUND.

quality of modifications using the JPlag software similarity checking and plagiarism detection software [4] as a third party tool. The plugin was tested on CodeChecker [5], a static analysis tool, whose developers provided their self-declared knowledge in the project's modules.

II. Related work

A. Developer Expertise

Ranking developers in a software project by their expertise in the source code – and on more abstract levels, in various programming technologies, such as languages and frameworks – is continuously relevant for companies and software projects. Expert identification is crucial in facilitating communication between programmers which, as a result, makes software development quicker and more effective. The time span of completing programming tasks has been identified as a key signal of a developer's competences, as Nguyen et. al [6] concluded it in their study. Robbes and Röthlisberger [7] have also found task completion time to be of great importance, however, they also relied on the time developers spent communicating with each other, using developer interaction repositories. Software repositories and static analysis are no new sources of data for programmer competences: Teyton et al. [8] developed a domain specific language to mine software repositories in search of project-specific developer competences.

Matter et. al [9] used their text-based developer identification to facilitate assigning bug tickets to the developer who most effectively can solve the problem. They mapped bug ticket description to the vocabulary used in developers' previous work descriptions. The same purpose was realized by Wu et al. [10] who applied other metrics like frequency, bug similarity and social network metrics.

B. Commit Classification

Research on classifying commits into various categories often relies on analyzing commit messages. Mauczka et al. [11] manually defined a vocabulary to classify commits as adaptive, corrective and perfective. A similar work has been done by Levin and Yehudai [12] who automated vocabulary expansion using machine-learning algorithms. They also refined the said 3 categories into several other, rather object-oriented classes. The textual approach was applied by Sabetta and Bezzi [13] but their research was focused on identifying security-related commits. They also treated modifications as text written in natural language, and used a small, targeted training data set for their machine-learning algorithm.

Levin and Yehudai's work was used as reference for Hönel et al. [14] who exceeded syntactic analysis by investigating the significance of code density in commit classification. Committing time has been identified in the work of Eyolfson et al. [15] as another possible factor in commit classification. Their research was focused on separating buggy and correct commits.

III. Expertise measurement

A. Background

Repositories, especially those of long-running projects tell plenty about development history and workflow. It creates an image of the gradual changes in the architectural design of the software. When it comes to maintenance and debugging, an emerging question can be "Who can I turn to for an explanation of the logic, structure and objective behind this code? Who is the expert in it?". Recognizing developers' knowledge in the project also supports task distribution, and helps planning quality assurance activities across the system lifecycle.

Developer expertise is intended to show a detailed picture of the code comprehension and expertise of the programmers in the project. If we consider smaller parts, developer expertise can be measured in the extent of comprehension of files, or even classes or functions. In our research, we quantify developer competence as a percentage value, which concerns one developer's knowledge of a single file based on their share of contribution. On a larger scale, it tells about each programmer's knowledge of the architecture and modules of a software which is the aggregation of their competence values of the corresponding files.

The way we measure competence via the activity done by the developer (editing the file) is based on the experiences on comprehension assessment [16, 17]. If one creates a new file, the creator developer's competence in that moment – and for some time after that – is 100%. If the file is modified by someone else, their competence may be reflected by the type and amount of changes they contributed.

Version control systems contain all the answers for the above questions in their commit history in an implicit way that requires meticulous analysis to be brought to the surface. Commits provide all the information about who is responsible for each line of code ever written for that repository in their blame data. We consider one commit as a unit in our calculation. The plugin parses a given number of commits from the commit history, which are then divided into *deltas* that represent the files modified by the commit author.

B. Data Parsing

The competence plugin consists of a *parser* and a *service* layer. The parser performs code analysis, repository mining, and competence calculation via software similarity checking. Fig. 1 shows the parser workflow. The service layer provides various tree-based diagrams of the data.

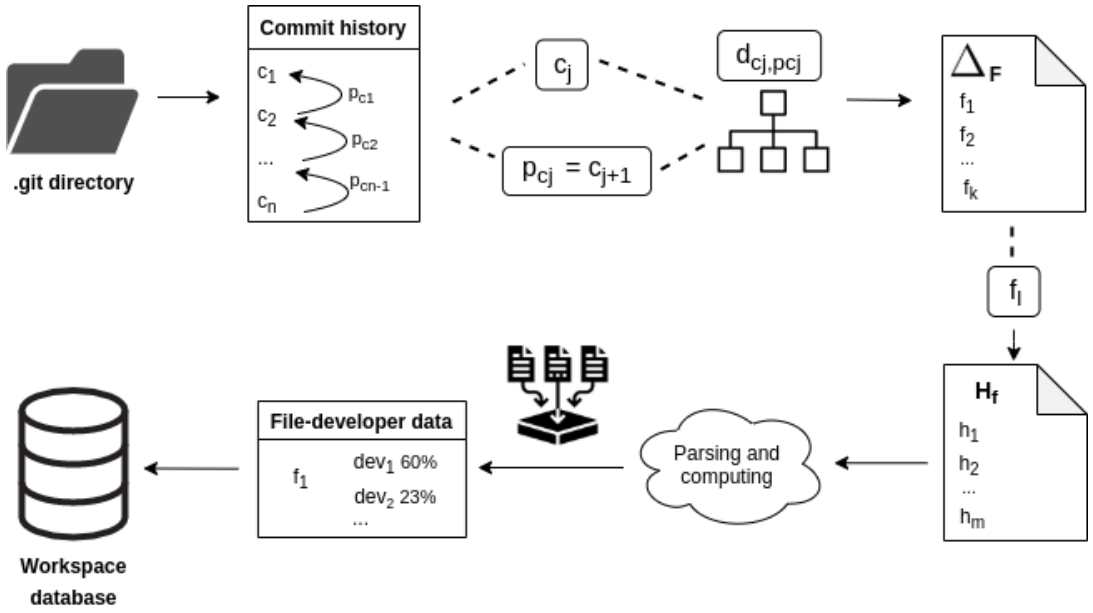

Figure 1. The methodology of commit history parsing. The commit history of a Git repository is parsed commit-by-commit. We take the diff tree (d_{cj},p_{cj}) of each consecutive pair of commits (c_j,p_{cj}), and analyze every file (f_k) that was modified.

The precondition of parsing is to have a Git repository. The parser also has to be provided with the number of commits *n* to be parsed as input parameter. The starting commit is the HEAD of the current branch. Once the repository is found, we need to collect the modifications done in the commits. We traverse through each commit in topological and chronological order, look at the list of modified files and calculate the quantity of modifications. As mentioned before, we use JPlag which applies token-based comparison to compare every file in a commit to its previous version [4]. JPlag provides a percentage of token similarity, this shows the significance of the modifications. A developer might have committed multiple times to a file, and the modified file content of commits might overlap in the file. In order to avoid overestimation, we consider the commit with the highest percentage value as the developer's expertise measurement.

It is important to mention that Git differentiates between the committer and the author of a commit, and uses two timestamps for authoring and committing. For example, when rebasing (squashing) multiple commits as part of a pull/merge request, the committer usually becomes the reviewer of the request, while the author is the original user who wrote the code. To avoid distortion of data caused by a different committer, we always consider the commit author and the author date in the calculation.

C. Commit Classification Methodology

JPlag provides a fair token-based comparison of two files. If we look at the percentage results, we may also classify the commits according to their significance. Commit classification is a well-researched topic, however, a great deal of research has been analyzing commit messages to identify the type of the commit [11, 12]. Classification is also a language-dependent task which makes automatic commit evaluation difficult. A software similarity checker is usually capable of processing multiple languages thus it means a practical solution for automated commit classification based on file content change. Its results are also more reliable in file by file evaluation where different types of modifications of different significance can be made in the same commit to each file.

Instead of classifying commits into various categories like the above mentioned research, our aim is determine whether a commit is *significant*, *partially significant*, or *insignificant*. We deem a commit significant, if all modifications are significant, partially significant if it contains at least one modified file with significant modifications, and insignificant if all modifications are insignificant based on software similarity checking. According to the work of Kawrykow [17], insignificant modifications might make up to 15.5% of the overall changes in a long-running software project. Such modifications require little to no understanding which is why we assort and ignore them in our calculation. They are usually included in refactoring commits, e.g. identifier renaming, adding, modifying or removing comments, reordering code without modifying its content, and any other refactoring that otherwise leaves the file semantically unchanged. In practice, this means modifications are ignored where the software similarity result is 100%.

The computation was executed on a file level granularity, then aggregated values for the directories were also calculated which provide a more easily interpretable result. Several languages, such as C++, Java, or C# support the logical organization of projects: each module is organized in its own directory.

IV. Results and Discussion

The methodology has been tested on CodeChecker [5], a static analysis tool built on the Clang Static Analyzer, developed by Ericsson and Eötvös Loránd University. It is

developed by a small group of constantly present programmers and occasionally by student developers and interns. CodeChecker is developed in multiple programming languages, mainly in Python, C++, and JavaScript, and other different file types occur such as JSON files, Makefiles, shell scripts, etc. It is a decent test project because of its many languages, reasonable amount of modules, and stable base of developers.

A. *Experimental results*

We analyzed the entire commit history of CodeChecker using the competence plugin. The history consists of more than 4 800 commits. The processed commits consisted of 13 674 file modifications in total, of which 3 263 (23.87%) were deemed insignificant based on the software similarity checking (JPlag returned a 100% similarity between the old and the new version of the file). These modifications were omitted from further processing. 334 commits (6.96% of the complete history) were categorized completely insignificant based, meaning all included file-level modifications were insignificant.

We also conducted a test by collecting the main modules of CodeChecker, and asked participants the following question: "To what extent do you know the source code of this module: *module name*?". 5 options were provided for each question:

(1) I have little to no knowledge of the source code.

(2) I know how the module works but know little of the actual source code.

(3) I have structural knowledge of the source code.

(4) I have a partially detailed knowledge of the source code.

(5) I have detailed knowledge of the source code.

The answer options correspond to a scale divided into equal sections: (1) corresponds to 0-20% of source code knowledge, (2) to 20-40%, (3) to 40-60%, (4) to 60-80% and (5) to 80-100%. The questionnaire was filled by 8 developers which might seem like a small number but related studies [18] show that the typical number of members in a project team is 9 developers.

Table I. shows the results of mapping the results of the plugin to the answers given by CodeChecker team members. The self-declared knowledge estimated by developers matched the results given by the plugin in 48.2% of the time which is a fair result. They overestimated their knowledge in 50% of the time, and underestimated in 1.8%. The high value of overestimation shows that developers on average have more knowledge of the source code than the commit history data shows. In order to be able to modify a file or a module, a programmer has to know more of the content than the actual code they contribute.

B. *Discussion*

As an answer to **RQ1**, software similarity checkers provide promising results in identifying and filtering out insignificant commits. However, the usage of binary categorization of commits – significant or insignificant -- provides an unsatisfactory result. A commit usually consists of several files with different levels of modifications. Therefore, all files cannot be treated homogenously. In order to provide more refined expertise values, we expanded our method to cover partially significant commits as well, with a more detailed, file-level granularity. Since similarity checking is evaluated file by file, we were able to compute overall expertise metrics without losing all information in a commit that contained at least one insignificant modification.

Regarding **RQ2**, using the defined, equally distributed scale in subsection A, the participants' answers deviated by 18.05% on average. This number signifies that the percentage scale should be recalculated to better fit the actual knowledge of developers as well as the "unseen" knowledge that is inevitable for contribution. The sections on the scale do not necessarily need to be of equal size, but further investigation is needed to define a more precise calibration. It is worth noting that answers deviate from plugin results mostly at the medium of the scale which means the calculation is most capable of showing the most and least competent persons in the given module. It is easier to match with the lowest parts of the scale since a developer presumably knows less about modules they haven't committed anything to at all than the ones they contributed to.

We have already discussed how we treat squashed commits in practice. However, the calculation entirely omits the committer who, in fact, might be possessing more knowledge than their commit history shows. Using the Git feature branching model or some similar type of development pattern is a common practice in software development teams. In such cases modifications are

TABLE I. QUESTIONNAIRE RESULTS

Team member	Matching answers	Overestimates	Underestimates
Product owner	3	11	0
Core developer 1	6	6	2
Core developer 2	11	3	0
Core developer 3	5	9	0
Non-core developer 1	7	7	0
Non-core developer 2	10	4	0
Non-core developer 3	11	3	0
Ex-developer	3	12	0
	54	**56**	**2**

Table 1. The results of our questionnaire conducted among the developers of CodeChecker. 8 team members answered the questions which were intended to collect their self-declared knowledge of each module. The participants had to classify their knowledge in the module's source code from 1 to 5, which was then mapped with their results of the competence plugin.

hardly ever committed to the main branch, they are rather written on separate branches. When a modification – let that be some refactoring, a bugfix, or an entirely new functionality – is done, the creator(s) make a pull request or merge request to integrate their new code into the main branch. It is also common practice that the author(s) of the new code cannot merge their modifications directly to the main branch, they have to wait for the review, and then the change requests or approval of other developers from the team [19]. The approved merge request is then usually merged by one of the reviewing developers [20, 21]. This means that reviewers have more knowledge of the source code than the amount they have authored. This might explain the overestimation of knowledge in parts of the code to which the overestimating developer has never even committed, or nearly not as much as their self-declared knowledge says.

The research shows that there is a strong correlation between a developer's extent of project-specific knowledge and their commit history, adding that the amount of contribution means some additional knowledge. The amount of self-declared knowledge and the results shown by the commit history are not directly proportional, further refinement might be needed to get accurate results from the percentage value.

The tendencies to over- or underestimate knowledge is interesting to observe in team members of various roles. A product owner might not commit as much as a developer but may often review other developers' code thus might actually know more of the project than their data shows. Similarly, a core developer usually contributes a lot to certain modules but may only review others, or direct the work of other developers. The plugin appears to give the most accurate results in non-core developers who may know some of the code in the modules they contributed to but are completely unaware of the others. The situation of an ex-developer is unclear: if they have been core developers, they could have been experts in particular modules without ever contributing to them due to the reasons mentioned above. In other cases, it is also possible that they overestimate their own expertise. The latter risk applies to any other developer in the project but the data provided by the ex-developer in our research is an outlier among the other participants. Answering **RQ2**, developers tend to give a higher estimate of their knowledge than the amount of their actual contribution shows but as we elaborated, various factors may influence the developers' perception of code-related knowledge.

V. Conclusion

In this research, we have developed the competence plugin, a static analysis tool which uses information obtained from version control repositories to mine developer-related information from the Git commit history. The plugin uses JPlag, a software similarity checker to execute token comparison on the modifications in a commit and determine their significance. The plugin computes a percentage value for each developer who have contributed to a file based on the committed modifications. This value can be used to draw an image of individual developer expertise in the project. The results of the plugin were tested on CodeChecker, and the results were compared to the self-declared knowledge of CodeChecker's developers. We found that the plugin provides promising results that fairly correspond to reality, considering the possible inaccuracy of a questionnaire that is based on self-declaration. The plugin is a helpful tool to better plan the usage of human resources, e.g. detecting possible knowledge loss due to strong developer fluctuation, or when source code related knowledge is unevenly distributed between developers.

The plugin will be further developed by considering additional developer-related metrics, such as roles in the project and the significance of merge request reviewing. The results given by the plugin are to be recalibrated to a more accurate, not evenly divided scale. We would also like to build on previous successful research in the field and expand our test projects.

References

[1] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, „Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report," in IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 170–181, 2017.

[2] A. A. Pandit, and T. Gauray, „Review of plagiarism detection technique in source code," International Conference on Intelligent Computing and Smart Communication 2019. Springer, Singapore, pp. 393–405, 2020.

[3] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás, „Codecompass: an open software comprehension framework for industrial usage," in Proceedings of the 26th Conference on Program Comprehension, pp. 361–369, 2018.

[4] L. Prechelt, G. Malpohl, M. Philippsen, et al, „Finding plagiarisms among a set of programs with JPlag," J. UCS8, 11, 1016, 2002.

[5] D. Krupp et al. CodeChecker – a static analysis infrastructure built on Clang Static Analyzer. https://github.com/Ericsson/codechecker

[6] T. T. Nguyen, T. N. Nguyen, E. Duesterwald, T. Klinger, and P. Santhanam, „Inferring developer expertise through defect analysis," in 34th International Conference on Software Engineering (ICSE). IEEE, pp. 1297–1300, 2012.

[7] R. Robbes and D. Röthlisberger, „Using developer interaction data to compare expertise metrics," in 10th Working Conference on Mining Software Repositories (MSR). IEEE, pp. 297–300, 2013.

[8] C. Teyton, M. Palyart, J.-R. Falleri, F. Morandat, and X. Blanc, „Automatic extraction of developer expertise," in Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. Pp. 1–10, 2014.

[9] D. Matter, A. Kuhn, and O. Nierstrasz, „Assigning bug reports using a vocabulary-based expertise model of developers," in 6th IEEE International Working Conference on Mining Software Repositories. IEEE, pp. 131–140, 2009.

[10] W. Wu, W. Zhang, Y. Yang, and Q. Wang, „Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in 18th Asia-Pacific Software Engineering Conference. IEEE, pp. 389–396, 2011.

[11] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, and T. Grechenig, „Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages," in International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 301–315.

[12] S. Levin and A. Yehudai, „Boosting automatic commit classification into maintenance activities by utilizing source code changes," in Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. Pp. 97–106., 2017.

[13] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 579–582.

[14] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, „Importance and aptitude of source code density for commit classification into maintenance activities," in 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 109–120, 2019.

[15] J. Eyolfson, L. Tan, and P. Lam, „Correlations between bugginess and time-based commit characteristics," Empirical [Software Engineering 19, 4, pp. 1009–1039., 2014.

[16] C.P.M. van der Vleuten, L.W.T. Schuwirth, F. Scheele, E.W. Driessen, and B. Hodges, „The assessment of professional competence: building blocks for theory development," Best Practice & Research Clinical Obstetrics & Gynaecology 24, 6, pp. 703 – 719., 2010.

[17] D. Kawrykow and M. P Robillard, „Non-essential changes in version histories," in 2011 33rd International Conference on Software Engineering (ICSE). IEEE, pp. 351–360., 2011.

[18] D. Rodríguez, M. A. Sicilia, E. García, and R. Harrison, „Empirical findings on team size and productivity in software development," Journal of Systems and Software, 85(3), pp. 562-570., 2012.

[19] A. Meneely, A. C. Rodriguez Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, „An empirical investigation of socio-technical code review metrics and security vulnerabilities," in Proceedings of the 6th International Workshop on Social Software Engineering. Pp. 37-44., 2014.

[20] Atlassian Tutorials: Making a Pull Request. https://www.atlassian.com/git/tutorials/making-a-pull-request

[21] GitHub Docs: Reviewing changes in pull requests. https://docs.github.com/en/github/collaborating-with-pull-requests/reviewing-changes-in-pull-requests