Poster: Understanding and Leveraging Developer Inexpertise

Lykes Claytor
Virginia Tech
frank8@vt.edu

Francisco Servant
Virginia Tech
fservant@vt.edu

ABSTRACT

Existing work in modeling developer expertise assumes that developers reflect their expertise in their contributions and that such expertise can be analyzed to provide support for developer tasks. However, developers also make contributions in which they reflect their inexpertise such as by making mistakes in their code. We refine the hypotheses of the expertise-identification literature by proposing developer inexpertise as a factor that should be modeled to automate support for developer tasks.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; **Expert search**;

KEYWORDS

expertise modeling, expert recommendation, history mining

ACM Reference format:
Lykes Claytor and Francisco Servant. 2018. Poster: Understanding and Leveraging Developer Inexpertise. In *Proceedings of ICSE '18 Companion, Gothenburg, Sweden, May 27-June 3, 2018,* 2 pages.
https://doi.org/10.1145/3183440.3195029

1 INTRODUCTION

Understanding the different topics for which individual developers have expertise allows software development teams to make informed decisions when assigning tasks or creating groups to work on software projects. Past work has looked into modeling developer expertise to support various software engineering tasks, *e.g.,* bug triaging [1], code review [2], or code collaboration [3].

Existing techniques for automated modeling of software development expertise share the assumption that every contribution that a developer makes to a software project demonstrates their expertise. Thus, these techniques analyze developer contributions to software in order to model the aspects of a software project that each developer knows well. However, sometimes developers make mistakes, and some of their contributions might actually contain evidence that they do not know a topic or aspect particularly well.

We propose the idea of modeling and analyzing developer *inexpertise* — the opposite of expertise. While modeling developer expertise allows us to understand which topics a developer knows well, we hypothesize that modeling developer inexpertise will enable us to highlight the topics for which developers still do not have a strong understanding, and for which they likely make mistakes.

Our proposed idea expands the hypotheses behind state-of-the-art techniques. Existing techniques assume that all developer contributions reflect expertise that developers gain from making such contributions. We refine this assumption by studying whether some developer contributions reflect *inexpertise*, *i.e.,* lack of expertise.

Developer-inexpertise models and analyses may enhance already-existing expert-recommendation techniques. Existing techniques model developer expertise according to how frequently developers modify or use files, classes, or terms in software. In existing models, developer contributions can only increase their expertise — not decrease it. Inexpertise data may be used to extend such models so that they decrease developer expertise according to the mistakes that they make. This new, extended model may refine existing techniques by allowing them to avoid the recommendation of developers for tasks for which they normally make mistakes.

More importantly, modeling developer inexpertise provides its own set of new applications. The ability to automatically determine gaps in developer knowledge has uses beyond improving expert-recommendation techniques. Inexpertise data could be used to recommend useful pieces of documentation or educational materials a software engineer might want to read in order to improve their skills. Additionally, new automatic techniques could flag contributions containing terms with which the author has previously shown inexpertise. Then, the development team would know that they should pay extra attention to such contributions, since they are more likely to contain a mistake than the average code contribution.

This paper motivates our idea of studying developer inexpertise, proposes some interesting research questions that this research area motivates, and discusses some applications of modeling and analyzing developer inexpertise.

2 RELATED WORK

McDonald and Ackerman [4] use the last change to determine the expert for a file. Mockus and Herbsleb [5] recommend the developer who made most changes to a file. Fritz *et al.* [3] consider additional factors such as frequency of code reading. Other techniques recommend developers to fix bugs represented by bug reports *e.g.,* [1, 7]), or to review code changes (*e.g.,* [2, 6]. Anvik *et al.* [1] recommend experts who have fixed similar bugs in the past Zanjani *et al.* [7] recommend experts who have interacted with the affected code.

Balachandran [2] recommends code reviewers based on the lines affected by the code change and nearby lines. Thongtanuam *et al.* [6] recommend reviewers that changed similar file paths to the reviewed change.

In contrast to past approaches to modeling developer expertise, we propose to model developer inexpertise — the concepts for which developers made mistakes — to feed automated techniques to support developer tasks.

3 MOTIVATING EXAMPLE

We describe a motivating example for a case in which it would be useful to automatically model and analyze developer inexpertise.

In this scenario, we have a software developer named Bob. Bob is a contributor to an open source project, and frequently makes contributions to a specific set of program modules. However, in some cases, Bob needs to write code that interacts with other parts of the program that deals with some domain-specific functionality, which he has not had many chances to study or contribute to. Due to Bob's gaps in this domain-specific knowledge, his contributions to these cross-cutting areas of the program often lead to defects that slow down the application and that cause issues for users. Since these defects are normally detected and studied after some time has passed, and since Bob also sometimes introduces defects in other areas of the program, he does not necessarily realize that his changes that affect the topics for which he does not have a strong expertise are significantly more error prone that his changes to other parts of the program.

In a scenario like this one, an automated technique to model and developer inexpertise would have been very useful to Bob and his team. Such a tool could be integrated with the IDE to warn Bob whenever he is working with code that deals with topics for which he does not have a strong-enough expertise, and for which he is more likely to make mistakes. Similarly, this technique could also warn the team that specific parts of the program have been affected by developer inexpertise and are more likely to contain defects. With this new understanding, developers would know that they should pay closer attention to the code that is more risky. Furthermore, this technique could also let developers know about terms and concepts for which they frequently make mistakes, so they can increase their training about them.

4 RESEARCH QUESTIONS AND PRELIMINARY FINDINGS

RQ1: Can we model the specific topics for which developers have inexpertise? A fundamental step in the area of developer inexpertise would be to be able to reliably model inexpertise. One potential way of modeling developer inexpertise separately from developer expertise would be from analyzing instances in which developers clearly made mistakes, *e.g.,* in fix-inducing code changes.

RQ2: Do developers make mistakes more often when working on their inexpertise topics? Once developer inexpertise has been modeled, we would like to study whether developers make significantly more mistakes when they are working with problems that involve the topics for which they have already demonstrated inexpertise. If this statement is true, then automated techniques could be built that made recommendations based on this trend.

RQ3: How does developer inexpertise change over time? Other studies have observed that developer expertise changes over time, *e.g.,* [3], since as time goes on developers can forget things and focus on new expertise areas. It would be beneficial to study whether and how developer inexpertise varies over time and, furthermore, whether time is a factor that could transform developer expertise into inexpertise.

Preliminary Findings. We have run some preliminary experiments over open source projects to start understanding the factors that demonstrate developer inexpertise. In our preliminary results, we succeeded in identifying sets of terms for individual developers for which they demonstrate their inexpertise, and which are different from the terms for which they demonstrate expertise. We also observed, that over time, developers tend to repeat the usage of some terms from one mistake to the next.

5 APPLICATIONS OF DEVELOPER INEXPERTISE

One clear direction is the enhancement of already-existing expert recommendation techniques.

Existing techniques could use an inexpertise model to lower the expertise score for developers that demonstrated inexpertise with the task that is currently being assigned. These techniques could also recommend developers in such a way that inexperienced ones are matched up with developers who can help educate them on the gaps in their expertise.

Inexpertise models could also enable additional applications, such as predicting when a developer is more likely to make a mistake. If a developer is writing a patch that uses terms or concepts with which they have previously demonstrated inexpertise, the IDE could warn them of their potential mistakes. Alternatively, the code could be flagged so that the developer team checks it more closely for defects. Finally, techniques based on inexpertise could also automatically identify developers who may have introduced a bug by comparing the bug report to developer inexpertise models.

Summing up, this unexplored avenue of research — automatically modeling and analyzing developer inexpertise — could lead to many valuable, high-impact applications.

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug?. In *International Conference on Software Engineering*. 361–370.

[2] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 931–940.

[3] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. 2010. A degree-of-knowledge model to capture source code familiarity. In *International Conference on Software Engineering*. 385–394.

[4] David W. McDonald and Mark S. Ackerman. 2000. Expertise recommender: a flexible recommendation system and architecture. In *Computer Supported Cooperative Work*. 231–240.

[5] Audris Mockus and James D. Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *International Conference on Software Engineering*. 503–512.

[6] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 141–150.

[7] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2015. Using developer-interaction trails to triage change requests. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 88–98.