tool-recommender-bot

Chris Brown and Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, NC
Email: dcbrow10@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Software contains a variety of useful tools and features designed to increase productivity that are often ignored or undiscovered by users. This lack of tool awareness can have dangerous consequences, especially in the software engineering industry. Automated recommendation systems have been created to software tool discovery, however these systems are not as effective as user-to-user recommendations. To improve the effectiveness of recommender systems, we implemented *tool-recommender-bot*, a novel system which allows toolsmiths to increase discovery of software engineering tools by integrating aspects of peer interactions, and found that our approach is useful in making recommendations and increasing awareness of a static analysis tool to programmers on 800+ GitHub projects.**

***Index Terms*—Software Engineering; Tool Recommendation; Tool Discovery; Open Source**

I. Introduction

Software is increasingly prevelant throughout society. To keep up with the increasing demand for technology, software quality has become an increasingly important metric for software development teams. Processes such as peer code reviews, unit tests, continuous integration, test automation, and more have been implemented to ensure software maintains a high quality.

Despite increased attention to the quality of code, the process of finding and fixing bugs in software, or debugging, is becoming more time-consuming and costly. A study by the National Institute of Standards and Technology reported that software engineers spend 70-80% of their time at work debugging, and on average it takes 17.4 hours to debug one error [20]. Additionally, these errors can cost companies millions of dollars. For example, a recent bug found in the cryptocurrency Ethereum resulted in the loss of $30 million.[1] Additionally, studies have shown the price of repairing these code failures becomes more expensive the longer the bug exists [3].

To make software engineers more effective and efficient in their work, many types of tools have been created to automatically perform code analysis, refactoring, security checks, and more for programmers. One such type is static analysis tools. Static analysis tools can improve software quality by automatically inspecting code without running the program. These tools can be useful for finding code defects early in the development process before code is released. According to a report from IBM, the relative cost for fixing errors also drastically increases the longer the fault exists in the code base through the software development lifecycle [5]. Previous studies also show that static analysis tools are effective in preventing bugs in code to save money for companies in addition to time and effort for developers [2].

Althouth quality is a primary concern for software producers and consumers, developers often ignore these useful tools which help improve the quality of their code [11]. There are many barriers to tool adoption, and one of the main reasons useful tools are ignored or underutilized is the discoverability barrier. This refers to when users are unaware of a tool's existence within software [17]. The tool discovery problem will continue to persist as applications become more "bloated" with features [16]. This lack of awareness of static analysis tools can lead to significant amounts of wasted time and money in the software industry.

To help solve the tool discovery problem, we developed *tool-recommender-bot* to automatically recommend useful tools to software developers. We designed our system to make suggestions by integrating characteristics of peer interactions found in previous work as well as concepts from software engineering. tool-recommender-botcontains a unique workflow that runs specific tools and makes customized recommendations to developers on GitHub[2], the most popular code hosting site for collaborative software development. This work seeks to answer the following research questions (RQs):

RQ1: How applicable is tool-recommender-bot to real-world software applications?
RQ2: How useful are recommendations from tool-recommender-bot to developers?

To answer these questions, we evaluate tool-recommender-botby recommending the Error Prone[3] static analysis tool on open source Java GitHub projects. We observed the frequency of recommendations made and examined how software developers reacted to receiving a suggestion from our system. This research makes the following contributions:

- a novel automated recommendation system tool-recommender-bot that allows researchers to recommend software development tools to GitHub developers, and

[1] https://qz.com/1034321/ethereum-hack-a-coding-error-led-to-30-million-in-ethereum-being-stolen/

[2] https://github.com

[3] http://errorprone.info

- an evaluation on the applicability and usefulness of our approach using ERROR PRONE

II. RELATED WORK

Our approach is based on previous research examining how users learn about and discover tools, the lack of tool adoption among software engineers, and existing automated tool recommendation systems.

Researchers have explored ways humans learn about new tools. Prior work in diffusion of innovations [21] and persuasion theory [23] show that the way messages are communicated impacts reception. There are a numerous methods for discovering tools in software, however, research suggests that recommendations from peers, or *peer interactions*, are the most effective. Murphy-Hill found that peer interactions were the most effective mode of tool discovery among software engineers compared to chance encounters, tutorials, descriptions, social media, or discussion threads [19] [18]. Similarly, Welty discovered that software users sought help from colleagues more than search engines and help menus [26]. Our work builds on the results of these studies by integrating qualities of effective peer-to-peer recommendations into an automated recommendation system.

Previous work has explored the tool adoption problem in software engineering and barriers preventing developers from adopting useful tools for important programming tasks. Researchers have created numerous tools to aid software engineers in their work, but these products are often ignored by developers [10]. Tilley and colleagues studied the challenges of adopting these research-of-the-shelf tools in industry [24]. Johnson and colleagues reported reasons why software engineers don't use static analysis tools to help find and prevent bugs in their code [11]. Xiao and colleagues examined barriers and social influences blocking developers from using security tools to detect and prevent vulnerabilities and malicious attacks [27]. Our project aims to increase software engineer tool adoption by developing tool-recommender-bot to automatically and effectively recommend useful tools to help complete development tasks.

Additionally, there are numerous existing technical approaches created to improve the tool discovery problem. Researchers have developed several active help systems using community involvement. Gordon and colleagues developed Codepourri, a system using crowdsourcing to collectively create Python tutorials [9]. Linton and colleagues designed a recommender system called OWL (organization-wide learning) to disseminate tool knowledge using logs throughout a company [14]. ToolBox was developed as a "community sensitive help system" by Maltzahn to recommend Unix commands [15]. Answer Garden helps users discover new tools based on common questions asked by colleagues [1]. SpyGlass automatically recommends tools to help users navigate code [25]

Fischer and colleagues examined systems that require users to explicitly seek help from the software, or passive help systems [7]. Fischer concluded technical passive recommendation systems, such as documentation and help menues, are ineffective and inefficient for users. Instead, active help systems are more useful for increasing tool awareness. We developed tool-recommender-bot to actively suggest useful programming tools and increase awareness for software engineers. Our approach differs from existing recommendation systems in the design and implementation of our tool.

III. TOOL

Our approach to improving tool discovery, tool-recommender-bot, aims to increase awareness and use of programming tools among software developers. This section describes the design and implementation of our system.s

A. Design

Previous research shows that recommendations between peers is an effective way to increase tool discovery and adoption [19]. Many existing help systems simulate user-to-user recommendations to increase awareness of application tools and features.

To better understand what makes peer interactions an effective mode of tool discovery, our prior work observed how colleagues recommend tools to each other while working on tasks. Our results found that *receptiveness* is a significant factor in determining the effectiveness of a tool recommendation, while other characteristics, such as politeness and persuasiveness, do not significantly impact the outcome [4]. We designed tool-recommender-bot to integrate user receptivity into our approach for making tool recommendations to increase awareness of programming tools.

Receptiveness

Previous work emphasizes the importance of receptivity. Fogg outlined best practicies for creating persuasive technology to change user behavior, and argued designers must choose a receptive audience [8]. Our prior work defined receptiveness using two criteria introduced by Fogg: 1) demonstrating a desire and 2) familiarity with the target behavior and technology. Below we explain how tool-recommender-bot was designed to recommend programming tools to software developers based on their desire and familiarity.

1. Desire

The primary desire of software users is to have enjoyable and problem-free experiences with software. Developers of these applications also have similar desires, to create high-quality and functioning programs for users. A 2002 study revealed that software engineers demonstrate this desire by spending the majority of the software development process and 70-80% of their time testing and debugging code [20]. To aid developers in finding, fixing, and preventing various issues in code, many different types of tools have been created to help accomplish these tasks. However, despite the existence of effective tools for detecting errors, the number of bugs in software is increasing [13]. We aim to increase awareness of these tools to improve software quality and developer productivity, ultimately meeting users' and developers' desire for less buggy software.

To target this desire of mistake-free code, our initial implementation of tool-recommender-bot automatically recommends the tool ERROR PRONE. ERROR PRONE is a static analysis tool created by Google to check for errors in Java code based on a suite of bug patterns. Static analysis tools like ERROR PRONE are useful for debugging and preventing errors in source code for applications, however they are often underutilized by software engineers [11].

2. Familiarity

Choosing an audience familiar with the target behavior is also vital to increasing adoption. To increase use of helpful programming tools, such as static analysis tools, our system focuses on making recommendations to software engineers within the context of the projects they develop. Familiarity with source code is important for creating software applications. Scalabrino and colleagues claim code understandability is one of the most important factors for software development, maintenance, debugging, and testing [22].

To choose a familiar audience, our approach makes recommendations on Github[4], a popular source code management and version control website that hosts millions of projects and serves millions of users. tool-recommender-bot makes its recommendations on pull requests, or proposed changes to source code submitted by programmers. Developers making these changes should be knowledgeable about the changes they propose as well as the code base to which they are contributing. Our approach suggests ERROR PRONE when a reported error is fixed by a developer in a pull requests but still exists elsewhere in the code to capitalize on their familiarity with the modifications and encourage the use of static analysis tools to find more bugs.

B. Implementation

tool-recommender-bot builds on four key concepts to automatically recommend tools to users and improve tool discovery based on our design goals for targeting developer receptivity.

1. Continuous Integration

Our system utilizes continuous integration to recommend useful tools before pull request changes are integrated into the main repository, or merged. tool-recommender-bot is implemented as a plugin for Jenkins, "the leading open source automation server" for source code deployment and delivery.[5] The system uses Jenkins to clone Github repositories and periodically check for newly-opened pull requests every 15 minutes. When a new pull request is found, our system uses Jenkins to automatically run our approach to recommend ERROR PRONE.

To analyze the source code, we target projects that use the Maven [6] build automation and software management tool for Java applications. Our approach uses Maven to automatically handle dependencies and perform the static analysis when the project builds. We inject ERROR PRONE as a Maven plugin to repository's *pom.xml* project object model file to add it to the build process. tool-recommender-bot then builds both the original version of the code before the proposed changes were made (base) and the changed version of the repository with the pull request modifications implemented (head) to inspect differences. Using Maven allows tool-recommender-bot to run on a large number of Java projects that use the popular build tool and also makes our approach extendable to recommend other tools implemented as Maven plugins in future work.

2. Fix Identification

After analyzing the base and head versions of the code, our approach parses the build output of each version to determine if any reported errors were fixed in the pull request. ERROR PRONE identifies faults found in the source code, and we developed an algorithm using that information to determine if changes made to the code in the head version fix the identified bug. Our technique uses the code differencing tool GumTree [6] to identify actions (addition, delete, insert, move, and update) performed between pull request versions and parse the code to convert the text into abstract syntax trees.

To determine if an error was fixed, we take several things into consideration: First, our approach ignores instances where only delete actions were detected between the base and head versions of a file. This avoids making recommendations in situations where bugs were removed but not necessarily fixed in refactoring tasks, such as deleting and moving code, renaming classes, etc. Second, we ignore occurrences of deprecated classes because, similarly, the error reported was not fixed but removed. Third, we do not consider error fixes that were made by changes to a different file because we want to make recommendations where the developer is familiar with the changes that occurred. These help us minimize the number of false positives and errant recommendations in our approach.

3. Fix Localizaton

When a fix is identified in the pull request, tool-recommender-bot then aims to find the location of the fix in the head version. To find the modified line that fixed a bug, we use GumTree to parse the Java file and convert it into abstract syntax trees. We look for the action closest to the offset of the error node calculated from the bug line number reported by ERROR PRONE. If the closest action is not a delete, then our approach take the location takes the location of that action. Otherwise, if the line was removed our algorithm searches for the closest sibling node or if none exists then the location of the parent. Additionally, the line of the fix had to be converted to the equivalent position in the pull request diff file displaying the changes between file versions, or number of lines below the "@@" header[7], before moving on to the next phase.

4. Code Review

[4]https://github.com

[5]https://jenkins.io/

[6]https://maven.apache.org/

[7]https://developer.github.com/v3/pulls/comments/#create-a-comment

Code reviews from co-workers are often standard practice in software development. Pull requests are the primary method of code contributions and code reviews on Github [28]. Our approach simulates peer reviews by making recommendations for static analysis tools as a comment to the pull request. tool-recommender-bot automatically runs ERROR PRONE and analyzes the code, providing feedback to developers based on their changes and the output of the tool. Github provides functionality for making comments at specific lines of code in a pull request, and tool-recommender-bot recommends ERROR PRONE as a comment at the fix location line from the previous step. Additionally, our system uses language similar to comments between co-workers in recommendations, such as using "Good job!" to compliment developers on their work [**?**].

To further increase adoption and use of static analysis tools, we only make a recommendation if ERROR PRONE reports other instances of the same error in the base version of the code. In the comment, tool-recommender-bot automatically adds direct links to at most two separate locations where the defect that was fixed still exists in the code. We hope this encourages the developer receiving the recommendation to use ERROR PRONE to fix similar errors and prevent more bugs from being merged into the master code base. Figure 1 presents an example recommendation from our system on a pull request review.

IV. METHODOLOGY

Our study evaluates the effectiveness of tool-recommender-bot by analyzing how often our approach makes tool recommendations and how developers respond to recommendations they receive from our system.

A. Projects

To determine the effectiveness of our approach, we used real-world open source software applications to evaluate tool-recommender-bot. Projects for our evaluation were selected from Github, which hosts millions of code repositories online. To narrow down the number of projects for our study, we chose repositories that met the following criteria:

- primarily written in Java,
- builds with Maven,
- and were ranked among the top 100 forked repositories for every month from February 2008 - December 2017.

ERROR PRONE currently only analyzes Java code, so our search was restricted to open source Java projects on GitHub. Our study focused on Maven projects because it is a popular build system for Java projects and also contains an ERROR PRONE plug-in. The current implementation of tool-recommender-bot automatically implements this plug-in by modifying the *pom.xml* file in the top directory. We selected the most forked repositories to collect popular projects with a lot of activity, and forking a repository is suggested as one of the GitHub best practices for contributing to a project[8]. Using a GitHub search API with the above constraints, we identified 917 projects for our evaluation.[9]

B. Study Design

We designed our evaluation to gather quantitative and qualitative data addressing our research questions.

RQ1

To answer our first research question, we observed how often our approach makes recommendations on newly opened pull requests. To analyze the 917 repositories simultaneously, we used Ansible[10] to install tool-recommender-bot as a Jenkins job on multiple virtual machines. In addition to the frequency of recommendations, we also tracked instances where ERROR PRONE defects were removed but not reported as fixed according to our fix identification algorithm in Section III.B.2, the number of occurrences where errors were fixed but no other instances of that bug were found in the code, and the false positive rate.

Johnson and colleagues discovered one of the primary barriers to static analysis tool usage among software engineers is false positives in the output [11]. To prevent unnecessary recommendations from our system, we manually reviewed all instances where tool-recommender-bot reported a recommendation should be made. After inspecting each reported pull request, we determined whether it was an appropriate case for our system to make a recommendation to developers. If so, we proceeded to post the comment recommending ERROR PRONE on the pull request. Otherwise, we noted the instance of a false positive in our approach and did not make a recommendation.

RQ2

To gather data on the usefulness of our system, we sent a follow-up survey to developers. Survey participants were users who received a recommendation from tool-recommender-bot on their pull request. We asked developers about their awareness of ERROR PRONE and how likely they are to use the tool in the future. The survey also included a free-response section to provide an opportunity for participants to add comments on the usefulness of the recommendation.

Developers voluntarily consented to complete the survey and provide feedback on our system. To ensure developers answered honestly, we notified respondents that their answers will be used for research purposes. Previous research has shown that survey participants are more motivated to answer truthfullly if they know they are contributing to research [12].

C. Data Analysis

We analyzed the data collected in our study to determine effectiveness of our automated tool recommendation system.

RQ1

Effective tool recommendation systems should have ample opportunities to regularly make recommendations to users.

[8]https://guides.github.com/activities/forking/

[9]https://docs.google.com/spreadsheets/d/1MRmVPiUeDlYMa0UODzMmFbGeYBbahzdeE
edit?usp=sharing

[10]https://www.ansible.com/

To determine how often tool-recommender-bot automatically recommends ERROR PRONE, we observed the number of pull request comments made by our tool. Our evaluation lasted for [some period of time], and we calculated the amount of true positive recommendations during that time span to measure the recommendation rate for each GitHub repository used in our study. To calculate the false positive rate, we compared the number of unnecessary instances where our tool proposed a recommendation found by researchers to the total number of instances where a recommendation reported by tool-recommender-bot.

RQ2

For our second research question, we accumulated responses from developers in our follow-up survey to determine the usefulness of our system. We utilized a five-point Likert scale for participants to rank how knowledgeable they were about the existence of ERROR PRONE before the recommendation and how likely they are to use ERROR PRONE for future development tasks. An optional free response section was provided at the end for respondents to describe explain why or why not they found the recommendation useful. These responses were used to analyze developers' reactions to our automated recommendation. Finally, researchers analyzed and independently coded open-ended responses from participants to further analyze the effectiveness of our system based on developer feedback.

V. RESULTS

A. How often can we expect tool-recommender-bot to make recommendations?

Tons of recommendations...

No false positives...

B. How useful are recommendations from tool-recommender-bot to developers?

Excellent responses from recommendees...

Something statistically significant...

VI. DISCUSSION

A. Observations

1) Why Were There So Few Recommendations?: Non-java changes, documentation, errors removed not fixed, error fixed everywhere in code...

B. Implications

Here's what our results say about improving tool recommendation systems...

VII. LIMITATIONS

Internal

An external threat to the validity of our study is that we only observed open source projects hosted on Github in our evaluation. Our results may not generalize to closed source software projects and their developers. To minimize this, we selected popular real-world software applications on Github owned by organizations to avoid the use of personal development projects. Additionally, our recommendation system has limited generalizablility due to the fact we currently only assess recommendations for the Error Prone static analysis tool on Java projects that build with Maven. Future work will look to extend tool-recommender-bot to include different types of tools, programming languages, and build systems.

VIII. FUTURE WORK

More programming languages (Python, C, C++ etc.)...

More types of tools to recommend (static analysis, security, debugging, etc.)...

More build systems (ant, gradle, TravisCI, bazel)...

IX. CONCLUSION

tool-recommender-bot is awesome

REFERENCES

[1] M. S. Ackerman and T. W. Malone. Answer garden: A tool for growing organizational memory. In *Proceedings of the ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems*, COCS '90, pages 31–39, New York, NY, USA, 1990. ACM.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[3] B. W. Boehm. Software engineering economics. *IEEE transactions on Software Engineering*, (1):4–21, 1984.

[4] C. Brown, J. Middleton, E. Sharma, and E. Murphy-Hill. How software users recommend tools to each other. In *Visual Languages and Human-Centric Computing*, 2017.

[5] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology & Planning*, 3(6):49–53, 2010.

[6] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[7] G. Fischer, A. Lemke, and T. Schwab. *Active help systems*, pages 115–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[8] B. Fogg. Creating persuasive technologies: An eight-step design process. In *Proceedings of the 4th International Conference on Persuasive Technology*, Persuasive '09, pages 44:1–44:6, New York, NY, USA, 2009. ACM.

[9] M. Gordon and P. J. Guo. Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 13–21, Oct 2015.

[10] V. Ivanov, A. Rogers, G. Succi, J. Yi, and V. Zorin. What do software engineers care about? gaps between research and practice. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 890–895. ACM, 2017.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[12] J. A. Krosnick. Response strategies for coping with the cognitive demands of attitude measures in surveys. *Applied cognitive psychology*, 5(3):213–236, 1991.

[13] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[14] F. Linton, D. Joy, H. peter Schaefer, and A. Charron. Owl: A recommender system for organization-wide learning, 2000.

[15] C. Maltzahn. Community help: Discovering tools and locating experts in a dynamic environment. In *Conference Companion on Human Factors in Computing Systems*, CHI '95, pages 260–261, New York, NY, USA, 1995. ACM.

[16] J. McGrenere and G. Moore. Are we all in the same "bloat"? In *Proceedings of the Graphics Interface 2000 Conference, May 15-17, 2000, Montr'eal, Qu'ebec, Canada*, pages 187–196, May 2000.

[17] E. Murphy-Hill. Continuous social screencasting to facilitate software tool discovery. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1317–1320, Piscataway, NJ, USA, 2012. IEEE Press.

[18] E. Murphy-Hill, D. Y. Lee, G. C. Murphy, and J. McGrenere. How do users discover new tools in software development and beyond? *Computer Supported Cooperative Work (CSCW)*, 24(5):389–422, 2015.

[19] E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 405–414, New York, NY, USA, 2011. ACM.

[20] S. Planning. The economic impacts of inadequate infrastructure for software testing. 2002.

[21] E. M. Rogers. *Diffusion of innovations*. Free Press, New York, NY, 5th edition, 2003.

[22] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: how far are we? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427. IEEE Press, 2017.

[23] L. Shen and E. Bigsby. The effects of message features: content, structure and style. *The SAGE handbook of persuasion developments in theory and practice*, 2012.

[24] S. Tilley, S. Huang, and T. Payne. On the challenges of adopting rots software. In *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering*, pages 3–6, 2003.

[25] P. Viriyakattiyaporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 27–41, Riverton, NJ, USA, 2010. IBM Corp.

[26] C. J. Welty. Usage of and satisfaction with online help vs. search engines for aid in software use. In *Proceedings of the 29th ACM International Conference on Design of Communication*, SIGDOC '11, pages 203–210, New York, NY, USA, 2011. ACM.

[27] S. Xiao, J. Witschey, and E. Murphy-Hill. Social influences on secure development tool adoption: Why security tools spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, pages 1095–1106, New York, NY, USA, 2014. ACM.

[28] Y. Yu, H. Wang, G. Yin, and C. X. Ling. Reviewer recommender of pull-requests in github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 609–612, Sept 2014.