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 developers have access to numerous tools designed to increase productivity by automatically performing development tasks, but these tools are often ignored or undiscovered by programmers. This lack of tool awareness can have dangerous consequences, especially in the software engineering industry. Automated recommendation systems were created to improve tool discovery, however these systems are not as effective as user-to-user recommendations. To increase awareness of helpful development tools, we introduce *tool-recommender-bot*, a novel system that integrates concepts of peer interactions and software engineering practices to increase tool discovery by effectively making recommendations to developers on open source GitHub projects.**

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

I. Introduction

Software is becoming more and more prevalent throughout our society. To keep up with the increasing demands for technology, software engineers are making efforts to emphasize —, or *software quality*. Throughout the software development life cycle, more focus is being put on metrics that impact both software producers and consumers such as functionality, reliability, usability, efficiency, maintainability, and portability [12]. However, despite increased attention to software quality, defects in code remains a persisting and escalating problem.

The 2017 Software Fail Watch by Tricentis, a software testing company, discovered that software failures impacted approximaltely half of the world's population (3.7 billion users) and caused $1.7 trillion in financial losses.[1] Additionally, the process of finding and fixing bugs in code, or debugging, is becoming increasingly time-consuming and costly. A study by the National Institute of Standards and Technology reported that software engineers spend 70-80% of their time debugging at work, and one error takes an average of 17.4 hours to debug [21]. Studies also show the cost of fixing an error becomes more expensive the longer a bug exists in source code [3], [5].

To improve code quality and prevent errors, researchers and toolsmiths have created numerous software engineering tools to aid developers in their work. These tools automatically perform a wide variety of software development tasks to save time and effort for programmers. Research shows that tools for static analysis [2], refactoring [18], security, code navigation, and more are beneficial for preventing errors and improving software quality. Ultimately, these useful tools can lead to reduced debugging costs and increased developer productivity.

Althouth quality is a primary concern for software engineers and users, developers rarely use tools designed to improve code [11]. There are many barriers to tool adoption, and one of the primary reasons useful tools are underused or ignored is the discoverability barrier, or when users are unaware of a tool's existence [17]. This lack of awareness for software engineering tools can lead to poor code quality, harmed users, and significant amounts of time and money spent fixing errors. While many automated approaches have been developed to increase knowledge of useful software tools and features, Murphy-Hill and colleagues found that learning about tools from colleagues during normal work activities, or *peer interactions*, is the most effective mode of tool discovery[20].

To help solve the tool discovery problem among software engineers, we developed *tool-recommender-bot*, a system to automatically recommend development tools. Our system makes recommendations by integrating characteristics of peer interactions and software engineering practices. tool-recommender-bot is designed for toolsmiths to increase awareness of their work by automatically running tools and making customized recommendations to developers on GitHub[2] projects. This paper 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 evaluated tool-recommender-bot by recommending Error Prone[3], an open source static analysis tool for Java code. We examined how often recommendations were made and how developers reacted to receiving suggestions from our system. This research makes the following contributions:

- a novel system for researchers to recommend software engineering tools on GitHub, and
- an evaluation on the effectiveness of tool-recommender-bot with Error Prone

[1] https://www.tricentis.com/software-fail-watch/

[2] https://github.com

[3] http://errorprone.info

II. Related Work

Our implementation of tool-recommender-bot builds on previous research examining tool discovery, lack of tool adoption among software engineers, and automated recommendation systems.

Researchers have explored how humans learn about new tools. There are various methods to discover tools in software, and research suggests recommendations between peers is the most effective way to increase tool awareness. Murphy-Hill found that peer interactions were the most effective mode of tool discovery compared to tool encounters, tutorials, descriptions, social media, and discussion threads [20] [19]. Similarly, Welty discovered that software users sought help from colleagues more often than search engines and help menus [27]. To improve the effectiveness of recommendations from our system, we integrate qualities of peer interactions into tool-recommender-bot.

Previous work has also explored the tool discovery problem and barriers preventing users from adopting new tools, specifically in the software engineering industry. 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 [25]. 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 [28]. Our project aims to increase tool discovery and adoption among developers by automatically recommending useful software enginering tools.

There are numerous existing technical approaches created to solve the tool discovery problem. Fischer and colleagues found that systems requiring users to explicitly seek help, or passive help systems, are ineffective and active help systems are more useful for increasing tool awareness [7]. Gordon and colleagues developed Codepourri, a system using crowdsourcing to make recommendations for Python code [9]. Linton and colleagues designed a recommender system called OWL (organization-wide learning) to disseminate tool knowledge using logs throughout a company [15]. ToolBox was developed as a "community sensitive help system" by Maltzahn to recommend Unix commands [16]. Answer Garden helps users discover new tools based on common questions asked by colleagues [1]. SpyGlass automatically recommends tools to help users navigate code [26]. 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. Peer Interactions

Previous research shows that recommendations between peers is an effective way to increase tool discovery and adoption [20]. 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.

a) 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 [21]. 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 [14]. 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].

1) 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 [23].

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. Software Engineering

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.

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.

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.

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.

Code Review

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 [29]. 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

[4]https://github.com
[5]https://jenkins.io/
[6]https://maven.apache.org/
[7]https://developer.github.com/v3/pulls/comments/#create-a-comment

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 system recommends software engineering tools and how developers respond to recommendations from our system.

A. Projects

We used real-world open source software applications to evaluate tool-recommender-bot. To choose study projects from the millions of GitHub repositories, we selected projects that met the following criteria:

- primariy written in Java,
- ranked a top 1000 monthly forked repositoy in the last 10 years,
- build using Maven, and
- do not already utilize ERROR PRONE.

We chose the most forked repositories to sample the most popular GitHub projects in terms of developer activity. Additionally, GitHub suggests forking repositories as a best practice for making contributions and pull requests to projects[8]. ERROR PRONE only analyzes Java code, so this evaluation was limited to Java projects. Maven is a build automation and dependency management system for Java projects. tool-recommender-bot requires Maven to automatically modify *pom.xml* files by adding the ERROR PRONE plug-in and running the static analysis tool. Using a GitHub search API with the above constraints, we identified 2003 projects to use for our evaluation.[9]

B. Study Design

We designed our evaluation to gather quantitative and qualitative data addressing our research questions. To analyze 2000+ repositories simultaneously, we used Ansible[10] to generate Jenkins jobs with tool-recommender-bot running on multiple virtual machines.

RQ1

To answer our first research question, we observed how often our approach makes recommendations on newly opened pull requests. 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 [13].

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. 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

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

[9]https://docs.google.com/spreadsheets/d/1MRmVPiUeDlYMa0UODzMmFbGeYPbzhzdeEv1YSgqC1oo/edit?usp=sharing

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

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] B. Kitchenham and S. L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE software*, 13(1):12–21, 1996.

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

[14] 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.

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

[16] 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.

[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 and A. P. Black. Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25(5):38–44, 2008.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.