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*—Recommendation systems were developed to improve the adoption of useful software tools and features designed to save time and effort in completing tasks that are often ignored by users. Previous research suggests that peer-to-peer recommendations is an effective mode of tool discovery, and the receptiveness of recommendees is a vital characteristic in determining the outcome of tool suggestions. To help increase awareness of useful tools, we developed and evaluated a new system tool-recommender-bot designed to integrate aspects of peer interactions and user receptivity into automated tool recommendations for software developers of real-world applications. Our findings suggest that tool-recommender-bot is awesome, cool, and very effective in improving tool discovery.**

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

I. Introduction

Software has permeated nearly every area of our 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 [19]. 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 [TODO: find a better example].[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 [16]. The tool discovery problem will continue to persist as applications become more "bloated" with features [15]. This lack of awareness of static analysis tools can lead to significant amounts of wasted time and money in the software industry.

To solve the static analysis tool discovery problem, we developed a new recommender system called *tool-recommender-bot*. tool-recommender-bot was designed to automatically recommend static analysis tools to software engineers. Previous research has shown that user-to-user recommendations are the most effective mode of increasing awareness of tools among software users [18]. We designed our system to make suggestions by integrating characteristics of peer interactions and concepts from the software engineering industry. To evaluate the effectiveness of our system, we studied the following research questions (RQs):

RQ1: How often can we expect tool-recommender-bot to make recommendations?
RQ2: How useful are recommendations from tool-recommender-bot to developers?

To answer these questions, we evaluated our system on open source Java projects to observe how many tool suggestions would be made based on past changes to the code base. We also examined how software developers reacted to receiving tool recommendations from our system. The main research contribution of our work is introducing the design and evaluation of a new automated recommendation system tool-recommender-bot, a novel approach to improving the discovery of static analysis tools for software engineers by simulating tool recommendations made between peers.

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

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 [20] and persuasion theory [22] 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 [18] [17]. Similarly, Welty discovered that software users sought help from colleagues more than search engines and help menus [25]. 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 [23]. 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 [26]. 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 [13]. ToolBox was developed as a "community sensitive help system" by Maltzahn to recommend Unix commands [14]. Answer Garden helps users discover new tools based on common questions asked by colleagues [1]. SpyGlass automatically recommends tools to help users navigate code [24]

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.

A. Design

Previous research shows that recommendations between peers is an effective way to increase tool discovery and adoption [18]. 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 [19]. 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 [12]. 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 ERROR PRONE.[2] ERROR PRONE is a static analysis tool created by Google to check for errors in Java code based on a suite of bug patterns. These tools are useful in debugging and creating desireable 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 [21].

To choose a familiar audience, our approach makes recommendations on Github[3], 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 reported errors are fixed by developers in pull requests to capitalize on their familiarity with their modifications. The GitHub interface allows users to review and comment on pull requests in situ with the proposed changes, this is where we make our recommendations.

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.[4] 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 [5] 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[6], before moving on to the next phase.

4. Code Review

Code reviews from co-workers are often standard practice in software development. Pull requests are the primary method

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

of code contributions and code reviews on Github [27]. 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 evaluation gathers both quantitative and qualitative data to evaulate the effectiveness of tool-recommender-bot.

A. Projects

To evaluate the effectiveness of our recommendation system, we assessed tool-recommender-bot on real-world open-source software applications. Github hosts millions of code repositories, and to narrow down projects for our evaluation we picked public repos that met the following criteria:

- primarily written in the Java programming language[7],
- build with Maven containing a *pom.xml* file in the top directory,
- and had at least 10 pull requests submitted by developers in the last six months (180 days).

Using the GitHub Search Repositories[8] functionality through the jcabi GitHub Java API[9], we were able to identify 106 projects that met the above criteria. A full list of the repositories used in our evaluation is available to view online.[10]

B. Participants

Are we collecting any demographic information on the developers?

[7]https://java.com
[8]https://developer.github.com/v3/search/#search-repositories
[9]http://github.jcabi.com/
[10]link

C. Study Design

We designed our study to address each of our research questions and address the influence of our recommendation system on improving tool discovery.

RQ1

To gather data on how often tool-recommender-bot makes recommendations, we counted the number of pull request comments made by our tool throughout the duration of our study. We wanted to determine how regularly our approach has opportunities to make recommendations to developers. For each project, we accumulated the number of recommendations made over [TODO: some period of time]. To run our evaluation on the 106 repositories simultaneously, we installed tool-recommender-bot as Jenkins jobs on multiple virtual machines to analyze the code bases in parallel.

In addition to the rate at which our system makes recommendations, we also collected other measurements to evaluate our system. We calculated our tool's false-positive rate, instances where errors were fixed but not found elsewhere in the code, so no recommendation was made, and occurrences where our approach reported an error was removed but not fixed according to our algorithm in Section III.B.2.

RQ2

To gather data on the usefulness of our system, we sent a follow-up survey to Github users who received a recommendation from our tool on a pull request they submitted. We asked the following questions to gather qualitative data:

- Question 1...

Developers voluntarily consented to participate in the survey and provide feedback on our recommendation.

D. Data Analysis

We analyzed the frequency of recommendations and follow-up survey responses to determine the effectiveness of our approach.

RQ1

To prevent false positives from our tool, all potential recommendations were redirected to the researchers before posting a comment on a pull request. Johnson discovered that one of the main barriers for underuse of static analysis tool among software engineers is reporting false positives in the tool output [11].

RQ2

Followed up with pull request authors to gather data on recommendation...

V. RESULTS

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

Tons of recommendations...

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

Excellent responses from recommendees...

Statistically significant data...

VI. Discussion

A. Observations

B. Implications

Here's what our results say about ways to improve 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 tools to recommend (static analysis, security, etc.)

More programming languages instead of just java...

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] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.