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 contains many tools and features designed to make users more efficient and effective in their work. However, despite a wide availability of such useful tools, many are often ignored or underutilized because users do not know of their existence [**?**]. The amount of tools provided by applications is also continuing to increase, making it more difficult for users to discover these helpful tools [**?**]. This lack of awareness leads to wasted time and billions of wasted dollars each year [**?**].

One area where worker effectiveness and efficiency is extremely important is the software industry. Software has permeated nearly every facet of our society and. Additionally, defects and errors in software cost companies millions of dollars to fix [**?**]. Quality is a primary concern for both software engineers and software users, however developers often ignore useful tools to help improve the quality of their code.

Automated recommendation systems can help solve this problem...

But existing recommendations systems are ineffective...

Peer interactions and receptiveness are effective [1]...

We created tool-recommender-bot to solve this... To evaluate the effectiveness of our approach, 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 and how software developers reacted to receiving recommendations. This research makes the following contributions:

- introduce the design and evaluation of tool-recommender-bot, a novel approach to improving tool discovery
- provide implications for designing effective automated recommendation systems.

II. Related Work

Improving tool discovery...

Existing automated tool recommendation systems...

Improving awareness of software engineering tools...

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 [6]. Many automated help systems are designed to simulate user-to-user recommendations in order to increase awareness of application tools and features, for instance Microsoft's intelligent office assistant Clippy[**?**].

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 [1]. 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 [3]. 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 [7]. 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 [5]. 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.[1] 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 can improve software quality by automatically inspecting code without running the program. These tools are useful in debugging and creating desireable applications, however they are often underutilized by software engineers [4].

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, and code understandability is one of the most important factors for software development, maintenance, debugging, and testing [8].

To choose a familiar audience, our approach makes recommendations on Github[2], 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, and 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.[3] 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 [4] 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 [2] 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

[1] http://errorprone.info
[2] https://github.com
[3] https://jenkins.io/
[4] https://maven.apache.org/

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.

4. 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 [9]. Our approach simulates peer reviews by making recommendations for static analysis tools as a comment to the pull request. 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 the adoption of static analysis tools, we report other instances of similar errors found by ERROR PRONE in the base version of the code. In the recommendation, tool-recommender-bot adds links to at most two locations where ERROR PRONE found the same error as the one that was fixed. We hope this encourags the use of ERROR PRONE to find more bugs in the pull request before the code is merged into the code base. Figure 1 presents a recommendation from our system on a pull request review.

IV. METHODOLOGY

A. Projects

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

- one of the top Trending projects[5] on Github based on activity by the community at the time of this writing,
- primarily written in the Java programming language[6],
- build with Maven,
- and are owned by a GitHub organization, instead of personal user account projects.

[5]https://github.com/trending

TABLE I
EVALUATION PROJECTS

Project	Java Files	LOC	Pull Requests

Details on projects used for study including GitHub repository name, number of Java files, lines of Java code, and total pull requests.

B. Study Design

We divided our study into two segments to address each research question:

1) RQ1: Last 100 pull requests on repositories...

2) 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...

No false positives...

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.

[6]https://java.com

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-botis awesome

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.