tool-recommender-bot

Anonymous Author(s)

ABSTRACT

To increase software engineer productivity, toolsmiths create tools and features to automatically complete software development tasks. However, these useful tools are often undiscovered or ignored by developers which is problematic for software applications that rely on programmer efficiency and correctness. This paper introduces *tool-recommender-bot*, an automated recommendation system created for researchers to increase awareness of their products by making recommendations to developers on online collaboration websites. To help improve tool adoption among software engineers, *tool-recommender-bot* uses a novel approach and original design principles to integrate concepts of user-to-user suggestions and industry practices in tool recommendations to developers, and our results suggest this approach is more effective in making tool recommendations to software developers on GitHub repositories.

KEYWORDS

Tool Recommendations; Tool Discovery;

ACM Reference format:
Anonymous Author(s). 2018. *tool-recommender-bot.* In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, Florida, United States, 4âĂŞ9 November, 2018 (ESEC/FSE 2018),* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To keep up with rising demands for technology, software engineers emphasize *software quality*. Throughout the software development life cycle, development teams monitor code metrics that impact expectations of software producers and consumers [13]. However, despite increased attention to software quality, buggy code remains a problem. The Software Fail Watch by Tricentis suggests software failures impacted 3.7 billion users and caused $1.7 trillion in financial losses in 2017.[1] Additionally, the process of finding and fixing bugs, or debugging, is a time-consuming and costly activity. The National Institute of Standards and Technology reports software engineers spend 70-80% of work time debugging and the average time to debug one error is 17.4 hours [22]. Fixing defects early in the development process is also important since studies show debugging costs increase the longer a bug remains in code [3, 5].

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE 2018, 4âĂŞ9 November, 2018, Lake Buena Vista, Florida, United States
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To improve code quality and prevent errors, researchers and toolsmiths have created software engineering tools to aid developers in their work. Prior work shows that tools for static analysis [2], refactoring [19], security, and more are beneficial for improving code and preventing bugs. These tools can automatically perform a wide variety of software development tasks to save time and effort for developers. Additionally, the Software Engineering Body of Knowledge recommends using development tools because they can be used to achieve "desirable characteristics of software products" [24]. Utilizing software enginering tools can reduce software errors and debugging costs while increasing developer productivity.

Although quality is a concern for developers and users, many helpful software engineering tools go unused by developers. One of the primary barriers to tool adoption is the discoverability barrier, where users are unaware of a tool's existence [18]. This lack of awareness for software engineering tools can lead to poor code quality, inconvenienced 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 developers prefer learning about tools from colleagues during normal work activities, or *peer interaction*, compared to other modes of tool discovery [21]. Similarly, Welty discovered that software users sought help from colleagues more often than search engines and help menus [27].

To help solve the tool discovery problem among software engineers, we developed *tool-recommender-bot*. Our system automatically recommends development tools by analyzing developer changes and integrating characteristics of peer interactions and software engineering practices. *tool-recommender-bot* is designed for toolsmiths to increase awareness of their products by systematically running their tools and making customized recommendations to developers, the largest code hosting platform in the world [10]. This paper presents the novel approach and design of our system and evaluates the effectiveness of our tool in a real-world scenario.

To evaluate *tool-recommender-bot*, we conducted a study making recommendations for Error Prone[2], an open source static analysis tool, to developers on the GitHub[3] code hosting platform. We calculated the effectiveness of *tool-recommender-bot* by measuring the frequency of recommendations and how developers reacted to receiving suggestions from our system. This research makes the following contributions:

- a novel approach for recommender systems to increase awareness of software engineering tools, and
- an evaluation of *tool-recommender-bot* recommending Error Prone to developers on GitHub.

2 RELATED WORK

Our implementation of *tool-recommender-bot* builds on previous research examining recommendation systems in software engineering, techniques for learning new tools and features, approaches

[2] http://errorprone.info
[3] https://github.com

for tool recommendations, and existing systems that recommend software engineering tools.

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 [11]. 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 [12]. 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 [16]. ToolBox was developed as a "community sensitive help system" by Maltzahn to recommend Unix commands [17]. 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.

3 APPROACH

tool-recommender-bot aims to increase discovery and adoption of useful software engineering tools by implementing three design pillars: *commend*, *apprehend*, and *recommend*.

3.1 Commend

3.2 Apprehend

3.3 Recommend

4 DESIGN

Previous research peer interactions influenced our three design pillars for *tool-recommender-bot*.

4.1 Peer Interactions

Previous research shows that recommendations between colleagues are the most effective way to increase tool discovery and adoption [21]. Murphy-Hill interviewed software engineers and found that user-to-user peer interactions occur less frequently in the workplace but are more effective than system-to-user solutions. [21] [20].

To better understand what makes peer interactions an effective mode of tool discovery, prior work by Brown and colleagues observed how software users recommend tools to each other while completing tasks. They 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 of a suggestion [4]. We designed *tool-recommender-bot* to integrate user receptivity into our approach for making tool recommendations to increase awareness of programming tools.

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

4.1.1 Desire. A key desire of software users is to have enjoyable and problem-free experiences with software. Producers of software applications also have a similar desire to create high-quality 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 [22]. 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 [15]. *tool-recommender-bot* aims to increase tool discovery by recommending software engineering tools designed to promote developers' desires for improving software quality.

Our goal is for *tool-recommender-bot* to increase awareness of software engineering tools. For our initial evaluation in this paper, we target software engineers' desire to produce mistake-free code by automatically recommending Error Prone. Error Prone is a static analysis tool that uses a suite of defined bug patterns to detect errors in Java code. 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 [12].

4.1.2 Familiarity. Choosing an audience familiar with the target behavior is also vital to increasing adoption. To increase use of helpful programming tools, our system focuses on making recommendations to software engineers within the context of the projects they develop. Scalabrino and colleagues claim code understandability is one of the most important factors for software development, maintenance, debugging, and testing [23]. *tool-recommender-bot* uses familiarity with source code to recommend tools to software engineers in code they modify.

To choose a familiar audience, our approach makes recommendations on proposed Github code changes submitted by users. Developers should be knowledgeable and familiar with the changes they propose, as well as the code base to which they are contributing. *tool-recommender-bot* suggests Error Prone when a reported error is fixed by a developer in a pull request or commit, and the same bug exists elsewhere in the code.

5 IMPLEMENTATION WALKTHROUGH

6 IMPLEMENTATION DESCRIPTION

6.1 Software Engineering

tool-recommender-bot incorporates key software engineering industry concepts to improve tool discovery among software developers.

6.1.1 Continuous Integration. Our system utilizes continuous integration to recommend useful tools beforecode 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 and commits every 15 minutes. When a new code change is found, our system uses Jenkins to automatically run our approach to recommend useful software engineering tools.

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 the desired software engineering tool 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 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. This also allows our system to easily be integrated into continuous integration builds for software applications to analyze code and recommend development tools to software engineers.

6.1.2 Debugging. 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 code modifications. The software engineering tool identifies faults in the source code, and we take the tool's output and use it in an algorithm we developed to determine if GitHub user code changes fix a reported bug. Our technique uses the code differencing tool GumTree [6] to identify actions (addition, delete, insert, move, and update) performed between altered code versions and parse the source 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 errors that are reported as fixed but were not located in files modified by the developer. This ensures that the GitHub user will be familiar with the code change and error fix. Second, we ignore instances where only delete actions were detected between the base and head versions of a file. This avoids making recommendations in situations where bugs are removed but not fixed. For example, a refactoring task such as renaming a class is presented in the GitHub diff as removing code from the original file. This can eliminate bugs reported by Error Prone between code versions, but may also just move the error to another location in the source code. Similarly, we also ignore deprecated classes because the error reported was not fixed. These conditions help us minimize the number of false positives and prevent errant recommendations to developers in our approach.

When a fix is identified in the changed version of code, *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 the software engineering tool, if applicable. 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 or commit diff file displaying the changes between file versions, or number of lines below the "@@" header[6], before moving on to the next phase.

6.1.3 Code Review. Code reviews from co-workers are often standard practice in software development [**?**]. Pull requests and commits are the primary methods of development contributions and code updates on GitHub [29]. Our approach simulates peer reviews by making recommendations for static analysis tools as a comment to the pull request or commit. *tool-recommender-bot* automatically runs software engineering tools and analyzes the code changes, providing feedback to developers based on their changes based on the output from the tool. Github allows users to make comments on specific lines of code changed in pull requests and commits. *tool-recommender-bot* leverages this functionality by recommending the tool as a comment at the fix location line from the previous step. Figure 1 presents an example recommendation from our system on a pull request.

To increase the likelihood of tool adoption, our system makes recommendations if the tool reports other instances of the fixed error in the latest version of the code. In the comment, *tool-recommender-bot* automatically adds direct links to at most two separate locations of the same defect. Our hope is that this encourages developers to use software engineering toolsin their work to fix the similar errors and prevent additional bugs. Additionally, our system uses language similar to a colleage in recommendations. For instance, *tool-recommender-bot* uses "Good job!" to compliment developers for fixing an error in their work [**?**].

7 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.

7.1 Projects

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

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

- primariy written in Java,
- build using Maven,
- do not already use Error Prone,
- ranked among the most popular and most recently updated repositories

Our evaluation was limited to Java projects since Error Prone can only analyze Java source code. To determine if a repository used Maven as a build system, we automatically checked if a Project Object Model (*pom.xml*) configuration file was located in the home directory. We also checked to make sure that the pom.xml did not already contain the Error Prone plugin to avoid projects that already use Error Prone. We selected projects that don't use Error Prone to increase awareness of the tool in recommendations. Developers are less likely to know about the tool if the projects they contribute to do not implement it in their build.

To get the most popular repositories, we filtered GitHub projects by the amount of stars based on numbers in the Fibonnaci sequence. We chose the most starred repositories to study the most popular projects on GitHub. Stars are a social aspect of GitHub where users can indicate their favorite projects and repositories of interest[7]. Using Fibonacci numbers allowed us to get a higher concentration of projects with a lower amount of stars, while fewer projects will have a very large number of stars. To filter of repositories, we grouped projects with 1 or 2 stars, 2 or 3 stars, between 3 and 5 stars, between 5 and 8 stars, etc., and sorted the top 100 projects in each group by when they were most recently updated. After using a GitHub search API to find projects that met the above criteria, we compiled a list of 789 code repositories. Out of those projects, one repository failed due to broken Unicode text. The projects used for our evaluation include a wide range of software applications providing a variety of services from large software companies such as Google and Apache to individual developers. A list of projects used for this study is publicly available online.[8]

7.2 Study Design

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

7.2.1 Setup. To analyze 700+ GitHub repositories simultaneously, we used Ansible[9] to generate Jenkins jobs running *tool-recommender-bot* on multiple virtual machines.

7.2.2 RQ1. To answer our first research question, we observed how often our approach makes recommendations on commits and 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 [12]. 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 repository modification, *tool-recommender-bot* determines whether it is an appropriate case to make a recommendation to the developer. For this study, we streamlined this process to send an email for the authors to review if a recommendation was proposed by our system. After the authors reviewed the code changes, if we deemed it was a true positive case we proceeded to use *tool-recommender-bot* to post the comment recommending Error Prone to the developer on the pull request or commit. Otherwise, we noted the instance of a false positive in our approach and did not make the recommendation. In situations where one repository code change provided multiple opportunities for a recommendation, the authors examined each of the changes and selected one of the errors reported as fixed to recommend to the developer.

7.2.3 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 or commit. 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 [14].

To futher examine the effectiveness of *tool-recommender-bot*, we compared our approach to sending email recommendations to developers. To study this, we found similar instances of code fixes by GitHub users where our system would recommend a tool and, instead of making the recommendation with *tool-recommender-bot* on the pull request or commit, send an email suggesting Error Prone to the developers. The emails also contained the recommendation feedback survey for developers, and we compared results to see how software engineers responded to receiving a recommendation by email vs. on GitHub from *tool-recommender-bot*. To send a recommendation via email, the developer must have an email address publicly available on the GitHub user profile.

7.3 Data Analysis

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

7.3.1 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 total number of new pull requests and commits, and compared it to the amount recommendations made by our tool. We calculated the rate of true positive recommendations during the span of our study to measure the recommendation rate for each GitHub repository used in our evaluation. To calculate the false positive rate, we compared the

[7] https://help.github.com/articles/about-stars/
[8] https://gist.github.com/tool-recommender-bot/1769ccd148508beabcd273a731723860
[9] https://www.ansible.com/

number of unnecessary instances where our tool proposed a recommendation found by the authors to the total number of instances where a recommendation reported by *tool-recommender-bot*.

7.3.2 RQ2. For our second research question, we accumulated responses from developers in our follow-up survey presented in recommendations from *tool-recommender-bot* and by email 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 approach based on feedback from software developers. We measured the response rate by observing the total number of recommendations made, the total number of survey responses, and the positive and negative responses from developers who received a recommendation via *tool-recommender-bot* and via email.

8 RESULTS

8.1 How often can we expect *tool-recommender-bot* to make recommendations?

Tons of recommendations...

No false positives...

8.2 How useful are recommendations from *tool-recommender-bot* to developers?

Excellent responses from recommendees...

Something statistically significant...

9 DISCUSSION

9.1 Observations

9.1.1 Why Were There So Few Recommendations? Non-java changes, number of errors removed but not fixed, number of errors fixed without another instance in the code, manual inspection of pull requests and commits...

9.2 Implications

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

10 LIMITATIONS

An internal threat to the validity of this work is our use of code differencing to determine if developers intended to fix a bug in a commit or pull request. We cannot definitively determine the intentions of GitHub developers making changes to a repository, however two authors analyzed the code changes and came to an agreement on if the modified patch was a fix before making a recommendation to the developer. Additionally, although we used a Likert scale to measure if GitHub users who received a recommendation were likely to use Error Prone in the future, we did not measure if the tool was actually adopted by the developers for future tasks.

Our evaluation has limited generalizablility due to the fact we only assessed recommendations for the Error Prone static analysis tool. This restricted our study to evaluate Java projects and a specific set of errors that can be reported by the tool. We selected Error Prone because it is able to report a wide variety of errors based on bug patterns for Java code. Another external threat to the validity of this study is the projects selected for our evaluation. We only examined open source repositories on GitHub, and these results may not generalize to developers of closed source software or projects on other code hosting sites, such as BitBucket.[10] To minimize this threat, we evaluated *tool-recommender-bot* on a large number of popular software applications on GitHub that provide many different services from a wide variety of software companies and developers.

11 FUTURE WORK

To improve discovery of software engineering tools, we plan to increase the practicality of *tool-recommender-bot* for researchers to implement our system with their projects. A next step is to extend our approach to work with multiple static analysis tools, such as Checkstyle[11] for Java. This will allow us to have more opportunities to make recommendations to developers based on the output from different tools. Additionally, we plan to update our system to work with software engineering tools that integrate as plugins for different build systems such as Gradle[12] and Travis CI[13], as opposed to just Maven projects. Future work will also extend *tool-recommender-bot* to work with different types of tools to increase adoption and usage, for example Find Security Bugs[14] which is useful for finding security vulnerabilities in Java web applications. Finally, future work will consist of expanding *tool-recommender-bot* to work with software engineering tools for different programming languages, such as the Pylint[15] static analysis tool for Python, clang[16] compiler and static analyzer for C and C++, and more.

12 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.

[10] http://bitbucket.org
[11] http://checkstyle.sourceforge.net/
[12] https://gradle.org/
[13] https://travis-ci.org/
[14] https://find-sec-bugs.github.io/
[15] https://www.pylint.org/
[16] https://clang.llvm.org/

[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] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM.

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

[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] I. C. Society, P. Bourque, and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[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.