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 a new approach to making software engineering tool recommendations that integrates characteristics from user-to-user suggestions and industry practices for researchers to increase awareness of their products among developers. To help improve tool adoption among software engineers, we implemented this approach in *tool-recommender-bot*, an automated recommendation system, and found our design is more effective for increasing tool discovery compared to other styles of tool recommendations.

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),* 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To maintain and meet rising demands for technology, software engineers emphasize *software quality* throughout the development process, monitoring metrics that impact software producers and consumers [13]. However, despite increased attention to quality, buggy code remains a problem as the number of software errors increases [15]. 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 reported software engineers spend 70-80% of work time debugging and on average one error takes 17.4 hours to debug [23]. Finding and fixing defects early during development is also important since studies show debugging costs increase the longer a bug remains in code [3, 5].

To improve code quality, researchers and toolsmiths have created software engineering tools to aid developers in their work. Research shows that tools for static analysis [2], refactoring [20], 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" [25]. Software enginering tools are useful for reducing software errors and debugging costs while increasing developer productivity.

Although software quality is important, development tools created to improve code are often ignored [10]. Previous work explores barriers preventing software engineers from adopting new tools: Tilley and colleagues studied challenges of adopting research-off-the-shelf tools in industry [26]; Johnson and colleagues reported reasons why software engineers don't use static analysis tools [11]; and Xiao and colleagues examined barriers of using security tools to prevent vulnerabilities and malicious attacks [29]. One of the main barriers to tool adoption is the discoverability barrier, where users are unaware tools exist [19]. Lack of software engineering tool usage can lead to poor code quality, inconvenienced users, and significant amounts of time and money spent fixing errors. Many existing automated approaches have been implemented to increase awareness of software tools and features, but Murphy-Hill and colleagues found that developers prefer learning about tools from colleagues during normal work activities, or *peer interaction* [22].

To help solve the tool discovery problem among software engineers, we developed an approach for making automated tool recommendations. Our approach integrates characteristics of peer interactions and software engineering practices, and can accomodate many different types of software engineering tools to make customized suggestions to developers. Our technique is based on three main design pillars- *commend*, *apprehend*, and *recommend*. These principles differentiate our approach from existing recommender methods and help improve the effectiveness of automated tool recommendations.

To evaluate our approach, we implemented *tool-recommender-bot*. Our initial implementation of *tool-recommender-bot* for this study recommends Error Prone[2], an open source static analysis tool for Java code, to developers on GitHub[3], a popular code hosting and collaboration website. We measured the effectiveness of our system by observing 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 implementation and evaluation of *tool-recommender-bot* recommending a static analysis tool to developers.

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

[2] http://errorprone.info

[3] https://github.com

ESEC/FSE 2018, 4âĂŞ9 November, 2018, Lake Buena Vista, Florida, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

This work builds on previous research examining recommendation systems in software engineering, techniques for learning new tools, approaches for tool recommendations, and existing automated recommendation systems that recommend software engineering 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 [18]. Answer Garden helps users discover new tools based on common questions asked by colleagues [1]. SpyGlass automatically recommends tools to help users navigate code [27]. 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 MOTIVATION

Our tool recommendation approach is motivated by previous research in tool discovery and peer interactions.

3.1 Peer Interactions

Research suggests recommendations between colleagues is the most effective mode of tool discovery. Murphy-Hill and colleagues interviewed software engineers and found that, even though they appear less frequently in the workplace, developers prefer peer interactions over system-to-user recommendations. [22] [21]. Similarly, Welty discovered that software users sought help from colleagues more often than search engines and help menus [28].

To better understand what makes peer interactions effective, Brown and colleagues observed how software users recommend tools to each other while completing tasks. Their results suggest *receptiveness* is a significant factor in determining the outcome of tool recommendations, while other characteristics such as politeness and persuasiveness are not as important [4]. Fogg argues a receptive audience is vital for designing persuasive technology, and describes receptiveness as: 1) demonstrating a desire and 2) familiarity with acquiring a target behavior [8]. In this research, our target behavior is the adoption of useful development tools and these two criteria for receptiveness influenced the design principles for our approach for recommending software engineering tools.

4 APPROACH

Our approach uses three design pillars to increase awareness of software engineering tools: *commend* developers on their work, *apprehend* additional opportunities for improvement, and *recommend* useful development tools to enhance the quality of the code.

4.1 Commend

Brown and colleages defined demonstrating a desire to adopt a particular behavior as users expressing interest in "discovering, using, or learning more information" about a new practice. In software engineering, developers desire to write high-quality programs without errors. This desire is demonstrated by the fact software engineers spend the majority their time testing and debugging programs [23]. Our approach capitalizes on developers' desires to produce mistake-free code by commending developers for their work. Futhermore, psychology research shows that people respond better to praise than blame [12]. To help improve tool adoption among software engineers, our approach analyzes code changes made by developers to determine their intentions, then compliments the developer on their attempt to improve the code quality.

4.2 Apprehend

Familiarity is also a key part of receptiveness and important in increasing adoption of a target behavior. Fogg suggests users are more likely to adopt a target behavior if they are familiar with it [8]. To improve usage of software engineering tools, our approach integrates familiarity. First, we make recommendations based on developers' contributions to recent projects. Scalabrino and colleagues claim code understandability is one of the most important factors for software development, maintenance, debugging, and testing [24]. Developers should be familiar with the project before making changes to the code base. More specifically, our approach apprehends similar pieces of code the developer may also want to change. After analyzing the code changes and commending the developer on their work, our approach analyzes the entire code base to search for similar opportunities for improvement.

4.3 Recommend

After integrating user receptiveness into our design, the final step is to make a recommendation.

5 IMPLEMENTATION WALKTHROUGH

To implement our approach, we developed *tool-recommender-bot* to recommend Error Prone to GitHub developers. This section gives an overview describing the general process of how the automated recommendation system works.

5.1 Commend

5.2 Apprehend

5.3 Recommend

6 IMPLEMENTATION DESCRIPTION

This sections provides technical details on the implementation of our approach in *tool-recommender-bot*. The design of our system is motivated by current practices in the software engineering industry to make *tool-recommender-bot* similar to a peer for developers.

6.1 Continuous Integration

To analyze developer changes, our system utilizes continuous integration concepts and tools to observe code modifications to GitHub repositories. *tool-recommender-bot* is implemented as a plugin for

Jenkins, "the leading open source automation server" for source code deployment and delivery.[4] We use Jenkins to periodically check for new modifications, commits and pull requests, to GitHub projects every 15 minutes. Our system ignores any commits or pull requests from developers that do not modify a Java file. When a new code change is found, Jenkins to automatically analyze the patch and run our approach.

To analyze the source code, we target projects that use the Maven [5] build automation and software dependency management tool for Java applications. We automatically inject Error Prone as a Maven plugin to a repository's project object model file (*pom.xml*) and run the build process with the tool. *tool-recommender-bot* 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 from the developer (head) to inspect differences. The base and head versions of the source code are tracked using the JGit Java API.[6]

6.2 Debugging

Before commending developers on their work, *tool-recommender-bot* must debug to find errors and determine if proposed code changes are a fix. After building the base and head versions with Error Prone, our system parses the output of each build to determine if any faults reported were removed between versions of code. To determine if a change fixes a defect, we developed an algorithm using the code differencing tool GumTree [6]. GumTree allows us to identify actions (addition, delete, insert, move, and update) performed between the altered versions of the project.

To determine if an error was fixed, we take several things into consideration: First, we ignore errors that are removed but were not located in a file modified by the developer. This ensures that the GitHub user will be familiar with the code changed and potential error fixed. Second, we ignore changes where only delete actions were detected between the base and head versions of a file. This avoids making recommendations in situations where defects were only removed by developers. For example, deleting a class will remove errors reported by Error Prone in the source code, however the intention was not to fix the bugs. Thus, for a change to be considered a fix there must be new code added or existing code modified by the developer. Similarly, we also ignore classes that are deprecated by developers. These conditions were put in place minimize false positives and prevent errant recommendations to software engineers in our approach.

When a fix is identified in the changed version of code, *tool-recommender-bot* finds the location of the fix in the head version. To find the modified line of code that fixed a bug, we use GumTree to parse the source code and convert it to abstract syntax trees. We look for the action closest to the offset of the error node determined from the line number reported by Error Prone. If the closest action is not a delete, then our approach uses the location of that action. Otherwise, our algorithm iteratively searches for the closest sibling node or parent nodes that is not a delete action. To apprehend different opportunities for similar changes, we iterate through the list of errors reported by Error Prone in the head version of code and look for instances of the same error that was fixed by the developer.

6.3 Code Reviews

Code reviews between developers are a standard procedure of software engineering teams to improve code quality [17]. This practice also applies to GitHub projects, with many repositories requiring approval from another developer before changes can be merged into the main code base [30]. Our approach simulates peer code reviews by making recommendations for static analysis tools as a comment on pull requests and commits. Github allows users to make comments on specific lines of code in situ with the code changes. *tool-recommender-bot* determines where to automatically make recommendation comments by converting the line number of the identified fix to the equivalent position in the diff file, or textual representation of code changes made in a commit or pull request, represented by the number of lines below the "@@" symbol in the header[7].

To increase the likelihood of tool adoption, *tool-recommender-bot* implements our approach by commending developers on their changes, apprehending chances for similar modifications, and recommending software engineering tools to find even more related errors. In the comment, *tool-recommender-bot* uses language similar to a peer to compliment the author's code contribution. For instance, our system uses "Good job!" to commend developers for fixing an error. Additionally, our system presents similar instances of the fixed error found elsewhere in the code. *tool-recommender-bot* automatically adds direct links to at most two different locations of the same defect where a similar fix can be applied. Finally, we recommend Error Prone and provide information about the tool to encourage developers to use software engineering tools in their future work to fix more related errors and more.

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:

- 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

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

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[8]. 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.[9]

7.2 Study Design

To evaluate the effectiveness of our approach, we compared *tool-recommender-bot* to different styles and mediums of tool recommendations

7.2.1 Setup. To analyze 700+ GitHub repositories simultaneously, we used Ansible[10] to generate Jenkins jobs running *tool-recommender-bot* on multiple virtual machines.

7.2.2 Quantitative. 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 [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 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 Qualitative. 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 Quantitative. 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 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 Qualitative. 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

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

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 Quantitative

Tons of recommendations...

No false positives...

8.2 Qualitative

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.[11] 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[12] 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[13] and Travis CI[14], 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[15] 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[16] static analysis tool for Python, clang[17] 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.

[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.

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

[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] W. A. Kennedy and H. C. Willcutt. Praise and blame as incentives. *Psychological Bulletin*, 62(5):323, 1964.

[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] L. MacLeod, M. Greiler, M. A. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: Understanding challenges and best practices. *IEEE Software*, PP(99):1–1, 2017.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.