Collaborative Bug Finding for Android Apps

Shin Hwei Tan
tansh3@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, Guangdong Province, China

Ziqiang Li
11510352@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, Guangdong Province, China

ABSTRACT

Many automated test generation techniques have been proposed for finding crashes in Android apps. Despite recent advancement in these approaches, a study shows that Android app developers prefer reading test cases written in natural language. Meanwhile, there exist redundancies in bug reports (written in natural language) across different apps that have not been previously reused. We propose *collaborative bug finding*, a novel approach that uses bugs in other similar apps to discover bugs in the app under test. We design three settings with varying degrees of interactions between programmers: (1) bugs from programmers who develop a different app, (2) bugs from manually searching for bug reports in GitHub repositories, (3) bugs from a bug recommendation system, Bugine. Our studies of the first two settings in a software testing course show that collaborative bug finding helps students who are novice Android app testers to discover 17 new bugs. As students admit that searching for relevant bug reports could be time-consuming, we introduce Bugine, an approach that automatically recommends relevant GitHub issues for a given app. Bugine uses (1) natural language processing to find GitHub issues that mention common UI components shared between the app under test and other apps in our database, and (2) a ranking algorithm to select GitHub issues that are of the best quality. Our results show that Bugine is able to find 34 new bugs. In total, collaborative bug finding helps us find 51 new bugs, in which eight have been confirmed and 11 have been fixed by the developers. These results confirm our intuition that our proposed technique is useful in discovering new bugs for Android apps.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Software maintenance tools**.

KEYWORDS

collaborative programming, test generation, recommendation system, Android apps

1 INTRODUCTION

Smartphones bundled with mobile applications have become indispensable. As mobile users rely heavily on their smartphones for their daily tasks, the reliability and usability of mobile applications are essential to ensure a satisfying user experience. According to a recent survey [1], 56% of respondents have experienced a problem when using a mobile application and 79% of them will retry an app once or twice if it fails to meet their expectations. Hence, there is a rising demand for testing and analysis techniques for mobile applications. Several automated techniques have been proposed for analysis [6, 11, 21, 28, 53, 73], testing [7, 8, 17, 48, 49, 56] and repair [63] of mobile apps. However, these automated techniques have not been widely adopted due to several reasons. Firstly, these techniques mainly focus on finding crashes and fail to find other types of bugs (e.g., UI-related bugs). Secondly, prior studies revealed that most app developers prefer manual testing compared to automated testing for finding bugs in their apps due to reasons such as lack of knowledge of testing tools, and learning curve of available tools [41, 44]. Thirdly, according to one of these studies [44], app developers prefer reading automatically generated tests written in natural language. However, existing automated testing techniques could only generate tests expressed in low-level events (e.g., using ADB commands) that are difficult for developers to understand.

For large software projects like Mozilla and Eclipse, there exist up to thousands of bug reports written in natural language. Hence, prior approaches on duplicate bug reports detection exploit the similarities between bug reports for bug localization [61, 70, 74]. In fact, many similar bug reports exist not only within the same project, but also across different software projects, especially for Android apps. Figure 1 shows two real world examples from ForkHub (a GitHub client)[1] where a developer who developed PocketHub and ForkHub found similar bugs in ForkHub by referring to issues in PocketHub. Meanwhile, Table 1 shows an extreme case where there exists a one-to-one correspondence in the GitHub issues between two apps of different categories (CameraColorPicker and Gnucash). However, there is little study on how to utilize the redundancies in bug reports across different Android apps for discovering new bugs.

Motivated by the testing needs of app developers and the redundancies in bug reports across different Android apps, we propose *collaborative bug finding*, a novel form of testing that exploits the similarities between Android apps for crafting test scenarios specialized for a given app under test. For Android apps, a *test scenario* includes (1) steps to reproduce, (2) test data (e.g., an image for image processing app), and (3) the expected behavior. The underlying
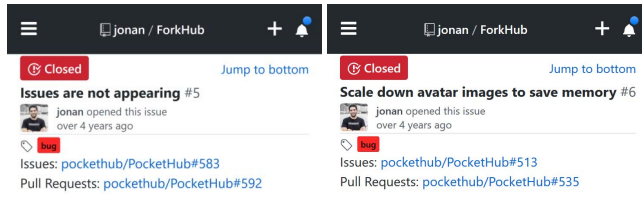
---

[1] https://github.com/jonan/ForkHub/issues/5, https://github.com/jonan/ForkHub/issues/6

Figure 1: App developers who developed PocketHub and ForkHub found similar bugs across two apps

Table 1: Example where there exists one-to-one correspondence for issues in CameraColorPicker (driver) and Gnucash (selected app)

Titles of issues in CameraColorPicker	Titles of issues in Gnucash
How do I get left top color.	When an account is edited, its color is lost.
publish to F-Droid	Publishing on F-Droid
Make gradlew executable	Make gradlew executable

assumption of our approach is that if App_A and App_B shared similar UI components and a developer A has found a bug Bug_A on App_A, then it is more likely to find a bug similar to Bug_A for App_B. To facilitate collaborative bug finding, we adapt the driver-navigator metaphor of pair programming by *emulating the role of a competent pair programmer via developers of other similar applications*. Specifically, we have designed three settings with *varying degrees of interactions* between the driver and the navigators. In the *coder-vs-coders* setting, the driver (developer A who develops App_A) drives the bug finding session by sharing his or her bug report R_A, while the navigators (other developers for App_B, ..., App_Z) read the bug report and think about whether the test scenario in R_A could be applied to App_B, ..., App_Z. In the *coder-vs-manual-issues* setting, the driver manually searches for bug reports from a different app App_B (navigator) and constructs a test scenario that is specialized for App_A. In the *coder-vs-auto-issues* setting, the driver provides App_A as a query and our issue recommendation system, BUGINE will automatically identify relevant GitHub issues by selecting issues from another similar app for App_A.

We introduce the concept of collaborative bug finding in a software testing course with 29 seniors (fourth year Computer Science students) where they cooperated through a group project that spans over ten weeks. We gathered both the opinions from students on the effectiveness of collaborative bug finding and the number of defects discovered. Both of these measurements show positive outcomes on the potential benefits of collaborative bug finding in an educational setting. Moreover, in some bug reports (e.g., in Figure 1), we also found evidence that Android app developers have been using the similarities between different apps for testing, which indicates the opportunities of employing collaborative bug finding beyond the classroom setting.

Overall, our contributions can be summarized as follows:

New Concept. We introduce the concept of collaborative bug finding, to the best of our knowledge, the first technique that exploits the fact that similar bugs appear even in different apps to craft specialized test scenarios for testing Android apps.

Improved teaching of software testing. Several techniques have been proposed in enhancing software testing education [18, 24, 25, 39, 43]. To the best of our knowledge, we present the first study
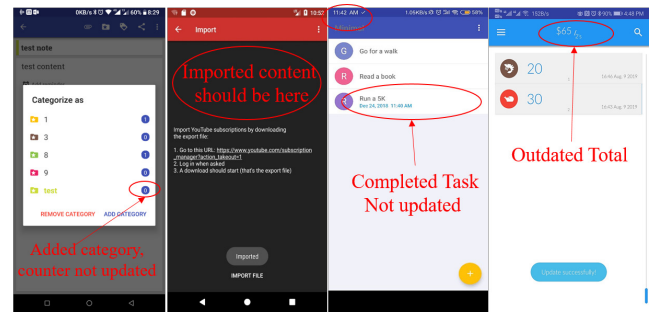

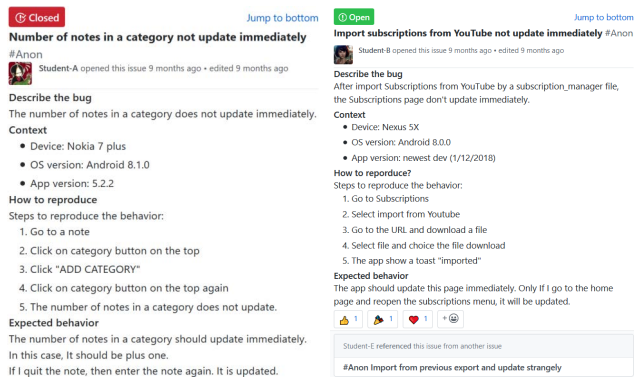**Figure 2: Four bugs (circled in red) found through collaborative bug finding across four apps.**

Figure 3: GitHub issue R_A reported by student S_A for Omni-Notes.

Figure 4: GitHub issue reported by student S_B for New-Pipe.

that leverages collaborative bug finding in GitHub classroom for teaching software testing course. Based on students' feedback of using GitHub classroom, we have reported one important feature to the developers of GitHub classroom and this feature has recently been planned for future release [60]. Moreover, our evaluation has demonstrated the effectiveness of our approach in improving teaching by helping students to find new bugs in Android apps.

New Recommendation System. We propose a new bug recommendation system, BUGINE for reducing the effort required for collaborative bug finding. Given an Android app A, BUGINE will automatically select relevant bug reports which can be used as tests for A.

Evaluation. We evaluate the effectiveness of collaborative bug finding in three settings in which different degrees of interaction between programmers are involved. In the coder-vs-coders and coder-vs-manual-issues setting, collaborative bug finding helps students discover 17 new bugs when evaluated in 20 apps. Meanwhile, BUGINE is able to recommend 34 new bugs for the five evaluated apps. In total, collaborative bug finding helps in the discovery of 51 new bugs, in which eight have been confirmed and 11 have been fixed by the developers. All the bugs found via BUGINE is publicly available at https://bugine.github.io/.

2 MOTIVATING EXAMPLE

We demonstrate the workflow of the coder-vs-coders setting of collaborative bug finding using four example apps and their bugs. Figure 2 shows the screenshots of the four open-source apps reported by four different students. The first app Omni-Notes is a note-taking app (Category: Productivity), while the second app New-Pipe is a media player app (Category: Multi-Media). Meanwhile, the third app Minimal To Do allows users to manage their to-do lists (Category: Productivity), whereas the fourth app CoCoin is an accounting app (Category: Finance). The process of collaborative bug finding proceeds as follows:

- Student S_A (i.e., the driver) found a bug in Omni-Notes that causes the counter representing the number of notes in the newly updated "test" category (circled in red) to be outdated when adding or removing a note. The user needs to restart the app for the increment to take place. Student S_A reported the GitHub issue R_A (as shown in Figure 3), and this issue has been confirmed and fixed by the developers [2].

- Student S_B read R_A and tried to derive a similar test scenario from R_A for New-Pipe app. Figure 4 shows the bug report filed by S_B [3]. Comparing Figure 3 and Figure 4, we infer that adapting the test scenario requires (1) creative thinking of the "steps to reproduce" in the context of New-Pipe, while two information can be reused from R_A: (2) the expected behavior of the app (i.e., the view should be updated), and (3) the test behavior after the incorrect view occurs (i.e., the view is updated correctly upon restart). With all the required information in mind, S_B found that although a message with "Imported" is shown when S_B tried to import subscriptions from YouTube, the content of the imported subscriptions are not displayed immediately, and could only be shown after an app restart. For this bug report R_B, the developer has recently added a pull request for fixing this bug.

- Student S_C tried to adapt the test scenario from R_A to Minimal To Do. In the adapted test scenario [4], the time reminder for a to-do was not updated immediately when the specified time for the to-do has passed, making the outdated time appeared in the screen. Similar to R_A, the view is updated only after restarting the app.

- Student S_D read R_A and realized that when the amount for lunch (the red icon) is modified from 45 to 30, the total displayed at the top of the screen ("65") was not updated accordingly despite showing a message "Update successfully"[5].

Although Figure 2 shows that all the four apps are relatively different and most of them are from different categories (except for Omni-Notes and Minimal To Do that belong to the same Productivity category), our example shows that similar bugs that are related to outdated view may exist across different apps. Apart from the four example bugs shown in Figure 2, another student S_E who also selected New-Pipe for testing had discovered similar problems when trying to import a previously exported subscription file (as shown at the end of Figure 4). Meanwhile, student S_A also encountered similar problem when modifying the tag of a note in Omni-Notes. In total, collaborative bug finding has helped in
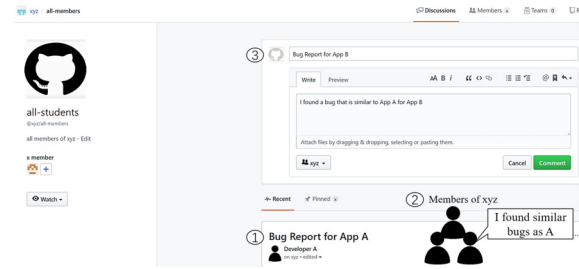
---

[2]https://github.com/federicoiosue/Omni-Notes/issues/625
[3]https://github.com/TeamNewPipe/NewPipe/issues/1919
[4]https://github.com/avjinder/Minimal-Todo/issues/113
[5]https://github.com/Nightonke/CoCoin/issues/47

Figure 5: Workflow for the coder-vs-coders setting
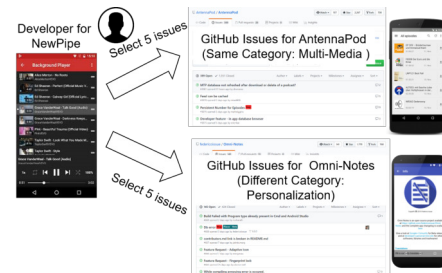

Figure 6: Workflow for the coder-vs-manual-issues setting

identifying five instances of similar bugs across four apps. By sharing a new GitHub issue R_A that student S_A found through manual testing, four students are inspired by R_A which helps them to derive specialized test scenarios for four new bugs in three different apps, respectively. It is worthwhile to mention that all these students are novice Android app testers who were given the task of testing apps that they have not used on a regular basis. This example *illustrates the effectiveness of collaborative bug finding in promoting creative thinking and the discovery of new bugs*.

Another interesting observation is that the problem of displaying outdated view when the values of a GUI component are modified seems to be a prevalent problem in Android apps. Our manual analysis of the fixes issued by the Omni-Notes developer and the New-Pipe developer indicates that this problem occurs due to inappropriate handling of asynchronous events, and could be solved by adding either observable sequences or background service to ensure that the view is being updated in real-time. As the fixes for the bugs in Omni-Notes and New-Pipe share similar high-level patterns but differ in the design choices (using either observable sequences or background service), this shows the potential of extracting bug-fix patterns from common issues for automated program repair [52, 65, 66]. More importantly, the fact that similar bugs could occur across four different apps and yet developers are willing to fix them promptly indicates *the importance and the prevalence of these bugs* in Android apps.

3 METHODOLOGY

We use the Research through Design (RtD) [27, 79] approach to design several settings to model different degrees of interactions between programmers. These settings aim to emulate different scenarios in GitHub where collaborative bug finding could be used.

3.1 GitHub and GitHub classroom

GitHub [6] is a social coding platform built on top of the git version control system and supports pull-based software development. *GitHub classroom* [36] is an open-source service launched by GitHub that allows instructors to use GitHub for teaching computer science courses. It includes several features (e.g., importing class roster, online discussions, and automatic creation of software repositories for individual and group assignments). Internally, it uses an organization to store course contents and student assignments. *GitHub team discussion*[7] is a feature in GitHub organization that supports communications among members within the same organization or the same team (a subset of members in an organization). We design our approach based on GitHub and GitHub classroom.

3.2 The coder-vs-coders setting

In the coder-vs-coders setting, we assume that both the driver and the navigators are programmers who belong to the same organization but each of them may develop a different app. A recent study of open-source Android apps in GitHub shows that the major contributors for an Android app also owned around five other repositories [14]. This suggests that our assumption (developers of different apps may belong to the same organization as other developers) generally holds.

Figure 5 shows the overall workflow for the coder-vs-coders setting. The collaborative bug finding process starts with a programmer A (the driver) posting a GitHub issue i for App_A in the team discussion page (Step ① in Figure 5). After reading i, another programmer within the same organization (the navigator) could read the GitHub issue and think about whether i is applicable for the app $App_{navigator}$ (in which $App_{navigator} \neq App_A$) (Step ② in Figure 5). This step requires the navigator to think creatively about whether there exist some "similarities" between App_A and $App_{navigator}$ that may allow i to be reproducible on $App_{navigator}$ (we leave it up to the navigator to define his or her own notion of similarities). If the navigator has successfully reproduced the issue in i, then he or she will post the new issue j to the team discussion (Step ③ in Figure 5). We call this a *pair sharing* with i being the originally shared issue and j being the derived issue. The process repeats when another member of the organization posts a new GitHub issue that is different from i and j. In this case, the greater the number of pair sharings for an issue, the greater the potential of discovering a general bug that is applicable to most Android apps (e.g., the outdated view bug in Section 2).

3.3 The coder-vs-manual-issues setting

In the coder-vs-coders setting, programmers need to wait passively for a new bug report. To improve the efficiency of collaborative bug finding, we design the coder-vs-manual-issues setting where programmers could get a one-on-one navigator for helping them find bugs in their apps. In this setting, the driver is a programmer who would like to perform collaborative bug finding but does not belong to any organization, whereas the navigators are open-source Android apps with a list of GitHub issues. This setting aims to

---

emulate the scenario where a novice app programmer would like to get advice for testing his or her newly created Android app.

Figure 6 shows the overall workflow for the coder-vs-manual-issues setting. Assume that a programmer A (driver) want to find bugs that may affect the app App_A that he or she developed. The process starts with A selecting randomly two apps: (1) an app App_B that belongs to the same category as App_A and (2) another app App_C that belongs to a different category than App_A. Selecting a same category app allows A to find specific bugs that may occur in apps that perform similar tasks, whereas selecting a different category app allows A to find general bugs that may occur across all Android apps. After the app selection, A chooses five GitHub issues (more issues could be selected if time allows) from App_B and App_C respectively. Then, A will think about whether the test scenarios in the selected issues could be reproduced in App_A. This step is similar to Step ② in Figure 5 where creative thinking is required.

4 DOES COLLABORATIVE BUG FINDING HELPS TO DISCOVER NEW BUGS?

Before evaluating the general applicability of collaborative bug finding for testing Android apps, we first study the general feasibility of collaborative bug finding as an approach that helps in constructing test cases for Android apps through different degrees of interactions. Our goal is to evaluate whether reading the bug reports written by others about another app (another app plays the role of a competent pair programmer) could lead to the creative thinking of similar testing scenario for the app under test and eventually lead to the discovery of new bugs for the app under test. **Experiment setup for the coder-vs-coders setting.** In a software testing course (CS409) with 29 students in Southern University of Science and Technology (SUSTech), we start the study by asking each student to form a team of 3–4. Each student is required to choose an app for testing from a list of open-source Android apps[8]. Note that although each student needs to test a different app within a team, we hope that the team setting will encourage collaboration among team members. We allow students to select their preferred app while providing some guidelines:

Ease of use: The project can be compiled successfully without errors, and can be executed in a device. This is compulsory.
Existing Tests: The project contains some test cases for validating the correctness of the app. This is compulsory.
Popularity: The app contains relatively large number of stars in GitHub or many downloads in Google Play.
Actively Maintained: There are recent commits to the projects.
Likelihood of finding new bugs: Projects with many bug reports may indicate that it is easier to find bugs in these projects but less likely to find new bugs (not duplicates of existing bugs).
Size: The project contains many Java classes that can be tested.

These guidelines aim to help students in judging the suitability of each app for testing. To ensure diversity of the selected apps, we also impose the rule that *only a maximum of three teams could select the same app and all apps selected by each member within a team must be distinct*. This selection results in 20 selected apps. For the coder-vs-coders setting, we encourage students to participate by sharing
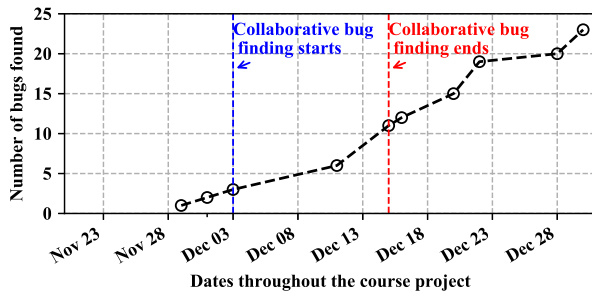

Figure 7: The total number of bugs found throughout the semester via coder-vs-manual-issues setting of collaborative bug finding

the bugs that they discovered in the organization discussion page (all students belong to the same organization). To encourage active participation, each student will receive 5 bonus points for each new bug reported to the app developer. At the end of the course, we manually analyzed all the issues posted in the discussion page, and attempt to answer two questions:

RQ1a: How many pair sharings exist in the discussion page throughout the entire course of the semester?

RQ1b: What are the types of bugs found through pair sharing?

Results for coder-vs-coders setting. Overall, 14 students have shared 29 GitHub issues for 11 open-source Android apps (the selected apps could be the same for some students). Among these 29 issues, we only observe five pair sharings. Meanwhile, the remaining bugs shared in the discussion page are discovered via other testing methods, which will be discussed in the subsequent paragraph. Interestingly, all the five pair sharings are related to the outdated view bugs described in Section 2.

> **Answer to RQ1:** There are five pair sharings in the discussion page. All of these bugs lead to outdated view.

Experiment setup for the coder-vs-manual-issues setting. To increase the number of discovered bugs via collaborative bug finding, we posted an assignment via GitHub Classroom that required students to write tests for their selected apps using two strategies: (Method 1) covering code changes for recent commit (we call this strategy *regression-inspired*), and (Method 2) the coder-vs-manual-issues setting of collaborative bug finding (refer to Section 3.3 more details). We compare these two strategies because both techniques rely on another similar app for constructing test cases (i.e., code changes relies on a different version of the same app, whereas collaborative bug finding uses a similar app as driver). For the regression-inspired strategy, students are required to write two test cases to verify the correctness of the changed code for the most recent commit. Note that we ask students to write two test cases so that they could design the tests to check both the normal behavior and the exceptional behavior. Using the coder-vs-manual-issues strategy, students are required to select five issues from an app within the same category, and five issues from a different category app (the category of an app is defined in this link [9]). For each selected issue *i*, students need to manually check whether they could

reproduce similar bugs in their selected apps based on *i* and write a new *derived issue* (issue that are inspired by *i*). To collect students' feedback for these two strategies, we asked students to answer the question: *"Which method do you think is more effective in finding new bugs? Why? Explain the reason in terms of efficiency (time taken) and effectiveness (likelihood of finding new bugs)"*. Our goal is to answer the following questions:

RQ2a What is the percentage of relevant GitHub issues when referring to the issues derived from same category app versus those from different category app?

RQ2b Compared to other testing approaches, how many bugs could collaborative bug finding discover? What are the types of bugs found?

RQ2c Considering students' feedback, what are the effectiveness and efficiency of collaborative bug finding versus constructing tests based on code changes?

RQ2a: Same category app versus different category app. We investigate whether the app of the same category or the app of a different category is more suitable as the driver app for collaborative bug finding. In total, there are 27 student submissions for this assignment (i.e., two students did not submit). These 27 students have selected 260 GitHub issues (131 issues from same category app, and 129 issues from different category app). Apart from selecting bug-related GitHub issues, we realized that students also select GitHub issues that include feature requests, build problems, questions, and documentation requests. For each issue, we classify its type and measure whether it is:

Definition 4.1. **Relevant** Given App_{query}, an issue *i* is relevant if similar functionality and steps to reproduce mentioned in *i* exist in App_{query} and does not lead to unexpected behavior.

Definition 4.2. **Reproducible** Given App_{query}, an issue *i* is reproducible if similar functionality and steps to reproduce mentioned in *i* exist in App_{query} and lead to unexpected behavior.

Table 2 shows the results of our manual inspection. The "Type of Issues" column in Table 2 represents the type of issues selected by students. Meanwhile, the "Relevance" column measures whether an issue is reproducible or relevant. The "Same Category" column and the "Diff. Category" column denotes the percentage of issues that are relevant to the app of the same category and the percentage of issues that are relevant to the app of a different category, respectively. Note that students could choose from apps that belong to 17 categories listed at [10]. For each type of issue, % reproducible + % relevant + % irrelevant (not shown in Table 2) = 100%. Overall, 21.18% of bug-related issues selected from same category apps could be reproduced in the selected apps, whereas only 11.96% of bug-related issues selected from apps of different category could be reproduced. Although selecting bug-related GitHub issues from same category app is more likely to be reproducible, our results show that *other types of issues (including features, build, question and documentation) from apps of different category could contain relevant information that may inspire the further improvement of other software artifacts.* Interestingly, students identified 10 derived issues that have been previously reported. In nine of these issues,

---

Table 2: The issue relevance comparison between app of same category and app of different category

| Types of Issues	Relevance	Same Category	Diff. Category
bug	reproducible	21.18%	11.96%
	relevant	32.94%	32.61%
feature	reproducible	33.33%	21.74%
	relevant	16.67%	21.74%
build	reproducible	20.00%	66.67%
	relevant	33.33%	0.00%
question	reproducible	20.00%	0.00%
	relevant	20.00%	0.00%
documentation	reproducible	50.00%	80.00%
	relevant	0.00%	0.00%

Figure 8: The number of new bugs discovered through different testing approaches across 11 apps.

Figure 9: Types of bugs found through collaborative bug finding.

we realized that the issues for the driver app and the navigator app are almost identical, which confirms with our assumption that similar bugs could occur across different apps. Meanwhile, in one of these derived issues of the AnExplorer app, a user mentioned that *"This problem exists in AnExplorer but not Amaze Explorer, however AnExplorer is required due to its other capabilities."*[11]. This example provides *concrete evidence of the potential usage of collaborative bug finding beyond the educational setting.*

> **Answer to RQ2a:** Among the selected bug-related issues, 21.18% of issues from same category app are reproducible, whereas only 11.96% of issues from different category apps are reproducible. However, selecting non-bug-related issues from different category apps may provide insights for future improvement of other software artifacts (e.g., build scripts, and documentation).

RQ2b: Number of bugs found and their types. Throughout the course, students also learn about writing tests based on several testing criteria (e.g., graph coverage based on manually drawn Event Flow Graphs and Input Domain Modeling (IDM) [9]). Meanwhile, Monkey[12], an automated tool for stress testing Android app, is the only automated testing tool taught in the course. Figure 8 shows the number of new bugs discovered for all the testing approaches. *In general, it may be difficult to keep track of the exact testing technique used to find a bug. However, this is not the case for collaborative bug finding because for each new bug found, we could refer to its original GitHub issue.* Overall, collaborative bug finding helps students in discovering 17 out of 29 new bugs. We also analyze how the accumulative total number of reported bugs in the discussion page evolves over time, including the key dates where we distributed the assignment and the deadline of the assignment. Figure 7 shows our analysis results. We can observe from Figure 7 that the total number of reported bugs increases significantly after distributing the assignment for collaborative bug finding. Specifically, Mann-Whitney U Test shows that the difference between the number of bugs found using our approach and the number bugs found using other approaches is statistically significant with $p < 0.05$. On the other hand, similar to prior study of the testing preference of Android developers [44], students use manual testing for finding five new bugs. Compared to manual testing approaches, Monkey only helps students in finding two new bugs. To understand the
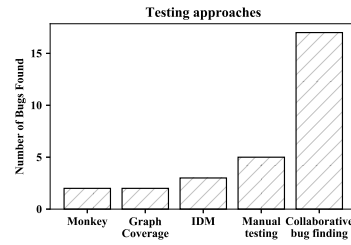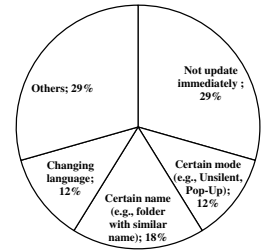
reasons behind the ineffectiveness of Monkey in the discovery of new crashes, we read the final reports written by students that summarized their learning experience and realized that students found three more bugs using Monkey but did not report them because (1) one crash becomes irreproducible when testing manually; and (2) most of the discovered crashes have been previously reported.

To understand the types of bugs that the first two settings help to discover, we carefully read all the 17 GitHub issues discovered via our approach. Specifically, we look for common keywords among all the GitHub issues. Figure 9 shows the results of our manual analysis. The results show that the update problem described in Section 2 is the most common problem (29%) among all the newly discovered bugs. Another common problem (18%) is that when certain names are used for creating a new file or a new folder, the app will exhibit unexpected behavior. For example, when the user of Amaze File Manager (file manager app) tried to create a folder with the same name as the current folder, it fails but the renaming should have been successful. In 12% of the reported bugs discovered through collaborative bug finding, when using some apps with different modes (e.g., non-silent mode and Pop-Up mode), the app behaves incorrectly. In 12% of the reported bugs, when app user tries to change the language either via the options in the app or open a file written in a non-English language, the displayed text is incorrect. For 29% bugs discovered via collaborative bug finding, they are non-crash related and specific to the selected apps.

> **Answer to RQ2b:** The coder-vs-manual-issues setting of collaborative bug finding helps students in the discovery of 12 new bugs. The improvement in the total number of bugs found is statistically significant. The bugs discovered via collaborative bug finding include: (1) prevalent problems affecting many apps (29% of them are related to the outdated view, 18% of them are related to the usage of specific names for file or folder creation, 12% of them are related to opening the app in specific mode, 12% are related to incorrect behavior with the change of language), and (2) specific problems that affect the selected apps (29%).

RQ2c: Comparison with regression-inspired approach. We manually read through the 25 answers for the question that compares (Method 1) regression-inspired approach, and (Method 2) collaborative bug finding. Figure 10 shows the students' feedback when comparing these two strategies. Overall, there are 16 students who prefer collaborative bug finding for the discovery of new bugs,
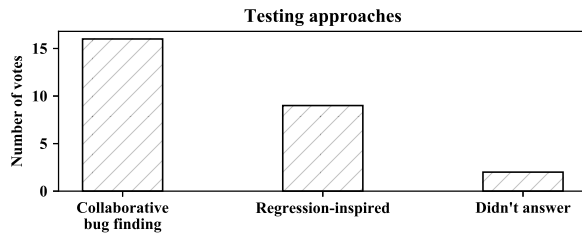
---

[11]https://github.com/1hakr/AnExplorer/issues/98
[12]https://developer.android.com/studio/test/monkey

Figure 10: Students' feedback on the comparison between collaborative bug finding and regression-inspired approach

whereas 9 students prefer the regression-inspired approach, and two students did not answer the question. For example, one student acknowledged the benefits of collaborative bug finding: *"Because many functions in the app in the same category are similar even totally same ... and others' report will also inspire the mind to find bugs which I never considered.".* Meanwhile, another student commented on the shortcoming of the coder-vs-manual-issues setting: *"method 2 takes more time to review different apps and search useful issues, while method 1 only needs to focus on small code changes. But method 2 is more likely to find new bugs".* In total, 15 students share similar concerns on the efficiency of collaborative bug finding. Based on general feedback that searching for relevant issues could be time-consuming, we design the coder-vs-auto-issues setting.

> **Answer to RQ2c:** 16 out of 25 students prefer collaborative bug finding over regression-inspired approach. However, 15 students admitted that searching for relevant issues could be time-consuming.

5 ISSUE RECOMMENDATION ALGORITHM

Based on the students' feedback on the weakness of the coder-vs-manual-issues setting, we propose BUGINE, an approach that automatically selects relevant GitHub issues for the app under test App_A so that developers for App_A could focus on the "creative thinking" step to derive specialized test scenarios. BUGINE's key steps include: (S1) using natural language processing to find similar apps by representing the common UI components of App_{query} and $App_{database}$ as app description files, (S2) automatically constructing queries from (S1) to select relevant issues, and (S3) ranking them based on their qualities.

Figure 11 shows the overall workflow of our issue recommendation system, BUGINE. Our approach first builds a database of GitHub issues obtained from open-source Android apps and pre-processes these issues to extract their metadata. For each app in our database, we extract its app description file for future comparison. Given an app under test App_{query}, BUGINE extracts its UI components to obtain its app description file. Then, we use the similarities between the app description file for App_{query} and the app description files for all the apps in our database to search for apps that are similar to App_{query}. This similarity is given as input to our ranking function, which prioritizes the GitHub issues in our database. Finally, BUGINE outputs a ranked list of relevant GitHub issues for App_{query}.
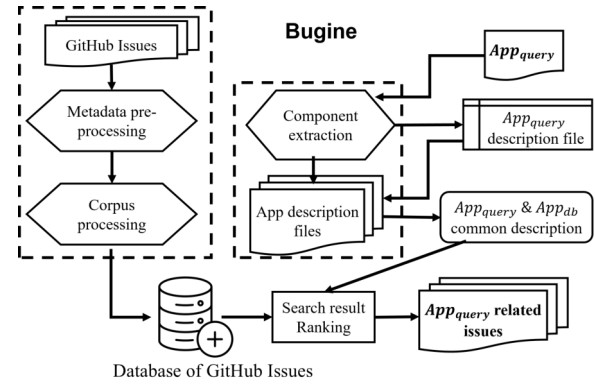

Figure 11: Workflow for our GitHub issues recommendation system

5.1 Building a database of GitHub issues.

To build our database, we first obtain a set of open-source Android apps by crawling GitHub. Our crawler selects an app A based on (1) the number of stars in GitHub, (2) the number of reviews and the number of downloads in Google Play, (3) the number of GitHub issues, and (4) its category. For each selected app, we extract all its issues and collected the metadata of each issue (e.g., title, author, number of user comments, labels, issue state, body, commit SHA, etc.). We also downloaded its source code from its master branch for subsequent steps. This results in a total of 23980 issues from 34 different applications that provide 10 different functionalities (e.g., cloud client, file explorer, web browser, notes, picture gallery, GitHub client, etc.).

5.2 Data Pre-Processing

As all the GitHub issues in our database are written in natural language, we use Natural Language Processing (NLP) techniques to pre-process them before storing them into our database. We also perform similar pre-processing for all the XML files. Specifically, we perform the following steps:

Tokenization: We convert each issue into lists of words (*tokens*).
Stopwords removal: Stopwords are commonly used words which do not have complete lexical meaning (e.g. "the", "is", "that"). We use the Python NLTK library [3] to remove stopwords.
Convention unification: Considering the naming convention of different Android UI components, BUGINE uses Humps [4], a python library that converts strings between snake case, camel case and pascal case, to unify the different naming conventions used in variable naming. We replace each underscore with white space and separate each composite word.
Stemming & lemmatization Stemming reduces inflected words to their base form, whereas lemmatization groups together the inflected forms of a word so that they can be analyzed as a single item. We use stemming and lemmatization for reducing inflectional forms of the parsed tokens.

After the data pre-processing, we convert the corpus into token streams, which will be used in the subsequent step.

Table 3: UI description patterns

Component	Description	Example	Extracted Names
Resource name	Resource name	android:id="@+id/**my_btn**"	my_btn
View name	The name for the type of UI component.	<**Button** android:id="@+id/my_btn" />	Button
XML file name	Layout name	**main_layout**.xml	main_layout

5.3 Extracting app description files.

The user interfaces of Android apps are commonly defined in XML layout files [2]. These files contain the definitions for the interface elements and their structures. Android *resources* are the additional files and static content used in the app code (e.g., bitmaps) [13]. Resources can be referenced via their resource IDs. The automatically generated file R.java contains all resources and their IDs.

In this step, BUGINE identifies and parses all the XML files within the folder src/main/res to generate app description files. We also discard element styles (e.g., colors and fonts) which are not essential to apps and retain only the important attributes that describe app behaviors. For each pair of apps (App_A, App_B), we compare and match their app description files. This description file is used for (1) classifying the category of the app (we assume that if the sets of UI components in App_A and App_B are similar, then App_A and App_B are of the same category), and (2) generating candidates search keywords for identifying the relevant GitHub issues. Then, BUGINE transforms these app descriptions into query phrases by summarizing the common parts in the two app description files. Table 3 shows the naming information in the XML files that we used to extract UI components that are common between two apps. This information helps us to convert for each XML file, each view and each resource in App_{query} to the query of the form:

$$XML\ file\ name \wedge View\ Name \wedge Resource\ name$$

Similarity Measures. We use two commonly used similarity measures for computing text similarities. To measure the similarities between the query and GitHub issue titles, we use *Overlap Coefficient* that computes the overlap between two finite sets X and Y [42]. Namely, Overlap Coefficient is defined as:

$$overlap(X, Y) = \frac{|X \cap Y|}{\min(|X|, |Y|)} \quad (1)$$

If X is the subset of Y or vice versa, then overlap(X, Y) is equal to 1. It ranges between [0, 1]. We choose Overlap Coefficient over the Jaccard Similarity [37] and Dice Similarity [22] because (1) the search query is usually shorter than the corpus in the database, and (2) it is sensitive to the size of the two sets.

To measure the similarities between the query and the text bodies of GitHub issues, we use the n-gram similarity because (1) it has been widely used in modeling natural language [62, 69], (2) Overlap Coefficient does not consider context (i.e., the surrounding words) but the text bodies usually contain detailed information and formal structural sentences. For the same reasons, we also use n-gram for calculating the UI components similarities. Specifically, we use the standard character-based n-gram from the Python NGram library [5] with the default value of n=3. To find items that are similar to a query q, the NGram library will split the query into n-grams,

---

[13]https://developer.android.com/guide/topics/resources/accessing-resources.html

Table 4: Factors used in ranking search results

Factor	Description
Issue length	Word count of issue body (int)
Issue status	Closed or opened (binary)
Ref commit SHA	Commit SHA referenced by issue (binary)
Issue reply num	The number of replies that an issue received (int)
Hit_all	Find all search keywords in the corpus (binary)
Hit_overlap	Overlap Coefficient between search keywords and corpus (float)
Hit_hot_words	Word count of descriptive hot words like *reproduce*, *defect* (int)

collect all items sharing at least one n-gram with q, and compute the similarity score based on the ratio of shared to unshared n-grams between strings.

We calculate the similarity between two corpora using:

$$g_{sim}(C_1, C_2, W) = \sum_{i=1}^{n} h(c_{1i}, c_{2i}) \times w_i \quad (2)$$

In this equation, $h(c_1, c_2)$ denotes the function used for comparing text similarities (e.g., Overlap Coefficient or n-gram), C_1 and C_2 are the pairs of corpora, c_{1i} and c_{2i} are the i-th part of the corpora, and W assigns a weight to each term.

5.4 Ranking Relevant GitHub Issues.

The goal of this step is to produce a ranked list of relevant GitHub issues for App_{query}. In previous steps, we obtain similar apps for App_{query}. We use each query phrase extracted from the app description files to search for related issues and ordered the results by their importance, relevance and reproducibility. We use several factors to rank the relevant issues. A key factor that determines the quality of a GitHub issue i is how well the test scenario has been described in i. An empirical study on Apache, Mozilla and Eclipse [80] shows that developers expect a good bug report to include *steps to reproduce*, *observed and expected behavior*, and *stack traces*. The study shows that a detailed bug report can help developers in narrowing down the search space for bugs and save time on fixing bugs. Based on the results of this study, we design the metrics used to evaluate the quality of each issue.

Table 4 shows the seven metrics that we used for ranking GitHub issues. Particularly, the Issue length measures how detailed an issue is (we assume that longer issue contains more information), whereas the Issue status depicts its importance (we assume that closed issues to be more important). The Ref commit SHA metric checks if the problem mentioned in issues has been fixed (as evidenced by the presence of a commit SHA). We include Issue reply num based on the assumption that issues with greater number of replies are more important. We use both Hit_all and Hit_overlap for finding the string similarities because we believe that if all the search keywords appear in the corpus (Hit_all), then the text should be given additional weights. The Hit_hot_words factor assigns a

higher score to GitHub issues with words that represent the characteristics of a good bug report [80].

Given an issue e, a component C that is similar to the search keywords, we calculate the ranking score $S(e, \vec{C}, W)$:

$$S(e, \vec{C}, W) = \sum_{i=1}^{n} f_i(e, \vec{C}) \times w_i \qquad (3)$$

where $f_i(e, \vec{C})$ denotes the value of factor f_i on issue e and search keywords C; and w_i denotes the weight for factor f_i. We perform a grid search to tune the weight for each factor.

6 EVALUATION

To the best of our knowledge, BUGINE is the first approach that ranks GitHub issues for Android apps. The approach closest to BUGINE is the advanced search feature in GitHub that can be used to narrow down the search for issues related to Android apps and filter the issue status, the number of comments, the date of the last update, and the label of each issue. However, there is no direct method to (1) filter issues from Android apps, (2) search for the UI components for an app nor (3) select the category of an app. Hence, we could not find any suitable baseline approaches.

We perform evaluation on the effectiveness of BUGINE to address the following research questions:

RQ3a What is the overall performance of BUGINE in recommending relevant GitHub issues?
RQ3b How many real bugs can BUGINE find?

6.1 Experimental Setup

We evaluate BUGINE on five open-source Android apps. Table 5 lists information about the evaluated apps. The "#Download" and the "Rating" column denote the number of downloads and the rating in Google Play, respectively. We select these apps for evaluation because they are diverse in terms of app category, sizes (5–31K lines of codes), popularity (60–378 stars in GitHub), and the number of issues. Although zapp is not able for download in GooglePlay, we choose this app because it was the most recently updated app in GitHub with frequent releases, which indicates that it has been actively maintained by the developers.

We evaluate the overall performance of BUGINE using two measures used in prior evaluations of recommendation systems [75, 78]:

Prec@k measures the retrieval precision over the top k documents in the ranked list:

$$\text{Prec@k} = \frac{\# \ of \ relevant \ docs \ in \ top \ k}{k} \qquad (4)$$

We measure the precision at k = 5, 10, 20, 50.

Mean Reciprocal Rank (MRR) For each query q, the MRR measures the position $first_q$ of the first relevant document in the ranked list [68]:

$$\text{MRR} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{\text{first}_q} \qquad (5)$$

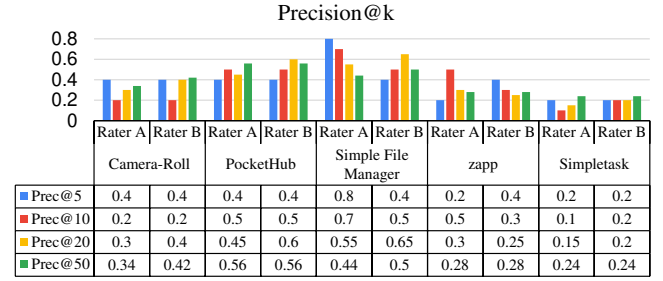The higher the MRR value, the better the ranking performance.
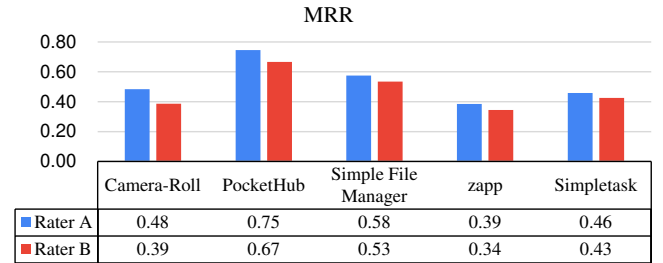

Figure 12: The Prec@k results for BUGINE

	Rater A	Rater B	Rater A	Rater B	Rater A	Rater B	Rater A	Rater B	Rater A	Rater B
	Camera-Roll		PocketHub		Simple File Manager		zapp		Simpletask	
Prec@5	0.4	0.4	0.4	0.4	0.8	0.4	0.2	0.4	0.2	0.2
Prec@10	0.2	0.2	0.5	0.5	0.7	0.5	0.5	0.3	0.1	0.2
Prec@20	0.3	0.4	0.45	0.6	0.55	0.65	0.3	0.25	0.15	0.2
Prec@50	0.34	0.42	0.56	0.56	0.44	0.5	0.28	0.28	0.24	0.24

Figure 13: The Mean Reciprocal Rank (MRR) results for BUGINE

	Camera-Roll	PocketHub	Simple File Manager	zapp	Simpletask
Rater A	0.48	0.75	0.58	0.39	0.46
Rater B	0.39	0.67	0.53	0.34	0.43

For *RQ3a* and *RQ3b*, *we use the same definitions of relevance as RQ2a (Def. 4.1 and Def. 4.2)*. For all the reproducible issues discovered, we reported them to the corresponding app developers.

All experiments were conducted on a machine with Intel (R) Core (TM) i7-8700 CPU @3.2 GHz and 32 GB RAM.

6.2 RQ3a: Ranking Performance of BUGINE.

Figure 12 shows the Prec@k results for BUGINE, whereas Figure 13 shows the MRR results for BUGINE. We include the computed values for the two raters (Rater A and Rater B) in both results for better comparison. The Prec@10 results range from 0.1 to 0.7, which means that among the top 10 issues recommended by BUGINE, there is at least one relevant issue. Meanwhile, the MRR values for BUGINE range from 0.34 to 0.75, which means that the ranking for the first relevant document ranges between 3rd (0.34) and 1st (0.75). Compared to the previous recommendation system that ranks source files for bug reports [75], we think that the MRR values are relatively high as their MRR values only range from 0.2 to 0.55. This indicates that BUGINE could recommend relevant issues for most of the evaluated apps (especially for PocketHub and Simple File Manager as their Prec@5 and MRR values are relatively high).

We observe that the Prec@k and the MRR results for SimpleTask are relatively low. This could be due to the fact that it is an app with limited number of features. Although it has the greatest number of GitHub issues among all evaluated apps (821), we realized that most of its GitHub issues are not bug-related (e.g., feature requests) with only 20% of these issues are marked as "bug". Another interesting observation is that the two reviewers (Rater A and Rater B) have different Prec@k values for Simple File Manager. We think that this difference is due to the fact that crafting specialized test

Table 5: Statistics and Evaluation Results of the Android apps used

App Name	Category	KLOC	#Downloads	Rating	Version No.	#GitHub Stars	#GitHub Issue (closed)	# Bugs Found (new,old)
Camera-Roll	Gallery	26.00	100,000+	4.2	1.0.6	420	227(133)	(11, 0)
PocketHub	GitHub client	31.35	10,000+	3.3	0.5.1	9429	644(526)	(12, 2)
Simple File Manager	Explorer	5.84	50,000+	4.5	6.3.4	378	189(130)	(6, 2)
Zapp	Broadcast	8.41	N.A.	N.A.	3.2.0	60	151(137)	(2, 7)
Simpletask	Reminder	24.80	10,000+	4.7	10.3.0	349	821(583)	(3, 2)

scenarios require creative thinking which is different across people. Nevertheless, the average inter-rater agreement is 0.76, which indicates substantial agreement between the two reviewers.

> **Answer to RQ3a:** The Prec@k and the MRR value show that Bugine could recommend relevant issues for all evaluated apps.

6.3 RQ3b: Number of bugs that Bugine finds.

Given an Bugine's issue, we evaluate RQ3b by manually replicating the issue to check if it is reproducible in the app under test and *we consider that Bugine discovers a bug if the issue is reproducible in the app under test.* The "# Bugs Found" column in Table 5 shows the number of bugs discovered by Bugine. In total, we found 34 new bugs and 13 old bugs in all the five evaluated apps. Compared to the coder-vs-coders and the coder-vs-manual-issues setting where 29 students find 17 new bugs when evaluated on 20 apps, Bugine could discover more bugs despite being evaluated only on five apps. This shows that the effectiveness of Bugine in recommending relevant issues, which leads to the discovery of new bugs.

Among all the evaluated apps, we are able to find the greatest number of new bugs in PocketHub. This result matches with the high Prec@k (Figure 12) and the MRR results (Figure 13). Moreover, the fact that PocketHub has the lowest rating among all evaluated apps indicates that many app users encountered bugs when using the app and the likelihood of finding new bugs is high.

> **Answer to RQ3b:** Bugine recommends 34 new bugs across five evaluated apps.

7 THREATS TO VALIDITY

External. Our evaluation results may not generalize beyond Android apps, the educational setting and testing approaches that we evaluated. To mitigate this threat, we used a large number of open-source Android apps. Furthermore, we also provide guidelines for students in their selection of the open-source apps to ensure the diversity of subjects in terms of popularity, ease of use, contains various test cases and a large number of reported GitHub issues. All participants of our study are fourth year CS students, and they are graded based on the number of new bugs discovered, which may lead to issues according to prior studies [15, 34]. We countered this threat by using bonus points to increase student participation [26] and reward the same bonus points to students regardless of the testing approaches used in finding the new bugs. To counter the threat of the general applicability of our approach beyond the classroom setting, we found some examples where developers of Android apps referred to other related apps when filing a new bug report.

Internal. We managed all the assignments via GitHub Classroom, and wrote scripts to automate the download of GitHub issues from the discussion page. During the manual inspection and classification of each bug, we had two undergraduate students (one of them being the author of the paper and another non-author) inspect the results independently to avoid the influence on the results. To avoid bias during the computation of the rank results for Bugine, we shuffled the rank results before presenting them to the two reviewers.

Construct. We used the number of newly discovered bugs to compare different testing approaches but other aspects (e.g., time taken to find bugs and the coverage of the generated tests) could be affected by using different testing approaches. We mitigate this threat by including and manually analyzing students' feedback on the potential benefits of collaborative bug finding.

8 RELATED WORK

Collaborative Programming. Pair programming is a form of collaborative programming where two people are working together on the same programming task [54]. Previous evaluations of pair programming show that the interaction between two programmers is effective in producing high quality software [13, 51, 71]. However, prior studies revealed that pair programming may not be cost-efficient [12, 19, 54]. Meanwhile, prior research focuses on the cooperative aspects of testing [46, 47, 72]. In collaborative testing for component-based systems, previous studies show that when several component-based systems use common components, testers of such systems can reduce testing costs and improve test effectiveness by sharing test artifacts [46, 47]. However, the proposed solutions rely on their own data sharing infrastructure for sharing test data across multiple component-based systems, which may be impractical. Similar to our approach that uses interactions for testing, the Timeline Tool leverages user interactions for failure reproduction [58]. Different from all these approaches, our approach uses different degrees of interactions between the driver and navigators for constructing test scenarios for Android apps.

Crowdsourcing. Prior work show promising results in using *crowdsourcing* [35] (a concept where various activities are outsourced to a group of people through online platforms) for performing several testing activities including mutation testing [59], usability testing [31, 45], GUI testing [23], and for education projects [16]. In the context of Android apps, Polariz combines crowdsourcing and search-based testing to enhance the activity coverage of generated tests [50]. Meanwhile, MoTiF uses crowdsourcing for collecting execution traces from users to reproduce context-sensitive crashes [30]. These approaches recruit crowd workers for testing several designated mobile applications. Although human-written

tests (collected from the crowd) have shown to be complementary to automatically generated tests, collaborative bug finding differs from these crowdsourcing-based approaches in that: (1) there is no direct interaction between crowd workers while writing tests in crowdsourcing, and (2) the crowdsourced tests are limited to particular apps and may not generalize to other untested apps.

GitHub. Prior research on GitHub focuses on the characteristics of the software repositories hosted on GitHub [20, 38], its transparency and collaboration [67], and its pull-based software development model [32, 33]. Several studies use GitHub as a collaborative learning platform for education [40, 76]. We present the first study that uses GitHub Classroom for teaching software testing.

Recommendation systems for Software Engineering. Several recommendation systems exist for performing software engineering tasks [10, 55, 57, 75, 78]. These systems aim to assist with fault localization based on ranking bug reports [55, 75, 78]. Different from these approaches, Bugine recommends GitHub issues for finding bugs in Android apps.

9 CONCLUSION

We propose collaborative bug finding, an approach that promotes the discovery of new bugs via different degrees of interactions between driver and navigators. In the coder-vs-coders setting, programmers within an organization communicate by sharing their newly reported GitHub issues. In the coder-vs-manual-issues setting, a developer for App_A selects several issues from same category app and from different category app to derive specialized test scenarios for App_A. Meanwhile, in the coder-vs-auto-issues setting, we introduce Bugine, an approach that automatically recommends relevant GitHub issues for the app under test. Our evaluation of all settings of collaborative bug finding shows that it helps in the discovery of new bugs. Specifically, it helps in finding five bugs via pair sharing in the coder-vs-coders setting. In the coder-vs-manual-issues setting, students reported 12 new bugs. Meanwhile, Bugine helps the discovery of 34 new bugs. The relatively high number of newly found bugs confirms that collaborative bug finding is a promising approach in testing Android apps.

We believe that this work opens several opportunities for future research:

Reusing bug reports from different apps. The concept of collaborative bug finding builds upon the idea of finding bugs by referring to bug reports of other similar apps. Although we have only applied this concept in the context of Android apps, it may be generalized to reusing bug reports for other types of applications with common characteristics (e.g., machine-learning applications).

Novel test generation approach. Instead of test input generation, *we reformulate the test generation problem as bug report recommendation problem.* As bug reports are written in natural language, our approach may help in relieving the burden of developers in learning a new android testing framework or APIs. Meanwhile, our evaluation shows that the types of bugs found via collaborative bug finding are mostly non-crash related, whereas existing automated testing approaches from Android apps focus on finding crashes. Moreover, prior testing techniques [29, 64] that extract specifications from code comments in natural language indicate potential benefits in improving automated testing with test oracles

obtained from bug reports. Hence, we believe that our approach is complementary to existing automated Android testing approaches. We leave as future work the investigation of whether collaborative bug finding can be combined with automated Android testing approaches to increase the number of bugs found.

Fully automatic collaborative testing. Automating collaborative bug finding involves: (1) test transfer which requires mapping all relevant UI components in the original issues to the derived issues (the app descriptions files in Bugine could be used), and (2) translating report in natural language to reproducible test scripts. Both of these steps are active research topics with promising initial results [77]. Collaborative bug finding recommends GitHub issues which are important prerequisite for these steps, and could spark future research in these topics.

Teaching software testing via GitHub Classroom. We present the first study that uses GitHub Classroom for teaching software testing. We have observed many benefits of using GitHub Classroom for distributing students' assignments and encouraging team collaborations, which are essential especially in the coder-vs-coders setting of collaborative bug finding. Moreover, as we have demonstrated the effectiveness of collaborative bug finding in helping students to find bugs in Android apps, we believe that the idea of using another similar application as a competitive "pair tester" could be potentially useful in other settings. In the future, it is worthwhile to investigate how to leverage GitHub and collaborative bug finding for teaching other software engineering courses or principles (e.g., test-driven development).

10 ACKNOWLEDGMENTS

REFERENCES

[1] 2016. Mobile Apps: What Consumers Really Need and Want A Global Study of Consumers Expectations and Experiences of Mobile Applications. https://docplayer.net/107560-Mobile-apps-what-consumers-really-need-and-want-a-global-study-of-consumers-expectations-and-experiences-of-mobile-applications.html. Accessed 2019-03-28.
[2] 2019. Layouts. https://developer.android.com/guide/topics/ui/declaring-layout. Accessed: 2019-02-28.
[3] 2019. Natural Language Toolkit — NLTK. http://www.nltk.org
[4] 2019. nficano/humps: Convert strings (and dictionary keys) between snake case, camel case and pascal case in Python. Inspired by Humps for Node. https://github.com/nficano/humps
[5] 2019. ngram 3.3.2. https://pypi.org/project/ngram/
[6] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. 2010. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 22.
[7] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 252–261.
[8] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.
[9] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.
[10] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. 2006. Supporting Online Problem-solving Communities with the Semantic Web. In *Proceedings of the 15th International Conference on World Wide Web*

(Edinburgh, Scotland) *(WWW '06)*. ACM, New York, NY, USA, 575–584. https://doi.org/10.1145/1135777.1135862

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[12] Andrew Begel and Nachiappan Nagappan. 2008. Pair programming: what's in it for me?. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 120–128.

[13] Jennifer Bevan, Linda Werner, and Charlie McDowell. 2002. Guidelines for the use of pair programming in a freshman programming class. In *Proceedings 15th Conference on Software Engineering Education and Training (CSEE&T 2002)*. IEEE, 100–107.

[14] John Businge, Moses Openja, David Kavaler, Engineer Bainomugisha, Foutse Khomh, and Vladimir Filkov. 2019. Studying Android App Popularity by Cross-Linking GitHub and Google Play Store.

[15] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. 2004. Issues in using students in empirical studies in software engineering education. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*. IEEE, 239–249.

[16] Zhenyu Chen and Bin Luo. 2014. Quasi-crowdsourcing testing for educational projects. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 272–275.

[17] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[18] Benjamin S Clegg, José Miguel Rojas, and Gordon Fraser. 2017. Teaching software testing concepts using a mutation testing game. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. IEEE, 33–36.

[19] Alistair Cockburn and Laurie Williams. 2001. Extreme Programming Examined. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Chapter The Costs and Benefits of Pair Programming, 223–243. http://dl.acm.org/citation.cfm?id=377517.377531

[20] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling exception handling bug hazards in Android based on GitHub and Google code issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 134–145.

[21] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. 2016. Exception handling bug hazards in Android. *Empirical Software Engineering* (2016), 1–41.

[22] Lee R Dice. 1945. Measures of the amount of ecologic association between species. *Ecology* 26, 3 (1945), 297–302.

[23] Eelco Dolstra, Raynor Vliegendhart, and Johan Pouwelse. 2013. Crowdsourcing gui tests. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 332–341.

[24] Stephen H Edwards. 2003. Teaching software testing: automatic grading meets test-first coding. In *Conference on Object Oriented Programming Systems Languages and Applications: Companion of the 18 th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Vol. 26. 318–319.

[25] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. 2007. Bug hunt: Making early software testing lessons engaging and affordable. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 688–697.

[26] Joseph R Ferrari and Stephanie McGowan. 2002. Using exam bonus points as incentive for research participation. *Teaching of Psychology* 29, 1 (2002), 29–32.

[27] Christopher Frayling. 1993. Research in art and design. (1993).

[28] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android Testing via Synthetic Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 419–429. https://doi.org/10.1145/3238147.3238225

[29] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/2931037.2931061

[30] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 88–99.

[31] Victor HM Gomide, Pedro A Valle, José O Ferreira, José RG Barbosa, Adson F Da Rocha, and TMGdA Barbosa. 2014. Affective crowdsourcing applied to usability testing. *International Journal of Computer Scienceand Information Technologies* 5, 1 (2014), 575–579.

[32] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[33] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: the contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 285–296.

[34] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering* 5, 3 (2000), 201–214.

[35] Jeff Howe. 2006. The rise of crowdsourcing. *Wired magazine* 14, 6 (2006), 1–4.

[36] Courtney Hsing and Vanessa Gennarelli. 2019. Using GitHub in the classroom predicts student learning outcomes and classroom experiences: Findings from a survey of students and teachers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 672–678.

[37] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37 (1901), 547–579.

[38] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[39] Cem Kaner and Sowmya Padmanabhan. 2007. Practice and transfer of learning in the teaching of software testing. In *20th Conference on Software Engineering Education & Training (CSEET'07)*. IEEE, 157–166.

[40] Csaba-Zoltán Kertész. 2015. Using GitHub in the classroom-a collaborative learning experience. In *2015 IEEE 21st International Symposium for Design and Technology in Electronic Packaging (SIITME)*. IEEE, 381–386.

[41] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[42] Gerald Kowalski. 2010. *Information retrieval architecture and algorithms*. Springer Science & Business Media.

[43] Daniel E Krutz, Samuel A Malachowsky, and Thomas Reichlmayr. 2014. Using a real world project in a software testing course. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 49–54.

[44] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622.

[45] Di Liu, Randolph G Bias, Matthew Lease, and Rebecca Kuipers. 2012. Crowdsourcing for usability testing. *Proceedings of the American Society for Information Science and Technology* 49, 1 (2012), 1–10.

[46] Teng Long, Ilchul Yoon, Atif Memon, Adam Porter, and Alan Sussman. 2014. Enabling collaborative testing across shared software components. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*. ACM, 55–64.

[47] Teng Long, Ilchul Yoon, Adam Porter, Atif Memon, and Alan Sussman. 2016. Coordinated Collaborative Testing of Shared Software Components. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 364–374.

[48] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[49] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[50] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 16–26. http://dl.acm.org/citation.cfm?id=3155562.3155569

[51] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The effects of pair-programming on performance in an ikrutz2014usinontroductory programming course. In *ACM SIGCSE Bulletin*, Vol. 34. ACM, 38–42.

[52] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article Article 15 (Oct. 2018), 37 pages. https://doi.org/10.1145/3241980

[53] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 33–44.

[54] Jerzy Nawrocki and Adam Wojciechowski. 2001. Experimental evaluation of pair programming. *European Software Control and Metrics (Escom)* (2001), 99–101.

[55] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A Topic-based Approach for Narrowing the Search

Space of Buggy Files from a Bug Report. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 263–272. https://doi.org/10.1109/ASE.2011.6100062

[56] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 300–311. https://doi.org/10.1109/ICSE.2017.35

[57] M. Robillard, R. Walker, and T. Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (July 2010), 80–86. https://doi.org/10.1109/MS.2009.161

[58] Tobias Roehm, Nigar Gurbanova, Bernd Bruegge, Christophe Joubert, and Walid Maalej. 2013. Monitoring user interactions for supporting failure reproduction. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 73–82.

[59] José Miguel Rojas, Thomas D White, Benjamin S Clegg, and Gordon Fraser. 2017. Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 677–688.

[60] stan6. 2018. Student should be able to view the deadline for an assignment. https://github.com/education/classroom/issues/1672.

[61] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 253–262.

[62] Ashish Sureka and Pankaj Jalote. 2010. Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*. IEEE, 366–374.

[63] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. IEEE, 187–198.

[64] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.

[65] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 471–482. http://dl.acm.org/citation.cfm?id=2818754.2818813

[66] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 727–738.

[67] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th international conference on Software engineering*. ACM, 356–366.

[68] Ellen M Voorhees et al. 1999. The TREC-8 question answering track report. In *Trec*, Vol. 99. Citeseer, 77–82.

[69] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 708–719.

[70] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*. ACM, 461–470.

[71] Laurie Williams, Robert R Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the case for pair programming. *IEEE software* 17, 4 (2000), 19–25.

[72] Tao Xie, Lu Zhang, Xusheng Xiao, Ying-Fei Xiong, and Dan Hao. 2014. Cooperative software testing and analysis: Advances and challenges. *Journal of Computer Science and Technology* 29, 4 (2014), 713–723.

[73] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[74] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 127–137.

[75] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 689–699. https://doi.org/10.1145/2635868.2635874

[76] Alexey Zagalsky, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. 2015. The emergence of github as a collaborative platform for education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, 1906–1917.

[77] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 128–139. https://doi.org/10.1109/ICSE.2019.00030

[78] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 14–24. http://dl.acm.org/citation.cfm?id=2337223.2337226

[79] John Zimmerman, Jodi Forlizzi, and Shelley Evenson. 2007. Research through design as a method for interaction design research in HCI. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 493–502.

[80] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.