Improving Feedback on GitHub Pull Requests: A Bots Approach

1st Zhewei Hu
Department of Computer Science
North Carolina State University
Raleigh, USA
zhu6@ncsu.edu

2nd Edward F. Gehringer
Department of Computer Science
North Carolina State University
Raleigh, USA
efg@ncsu.edu

***Abstract*—Rising enrollments make it difficult for instructors and teaching assistants to give adequate feedback on each student's work. Our course projects require students to submit GitHub pull requests as deliverables for their open-source software (OSS) projects. We have set up a static code analyzer and a continuous integration service on GitHub to help students check different aspects of the code. However, these tools have some limitations. In this paper, we discuss how we bypass the limitations of existing tools by implementing three Internet bots. These bots are either open source or free for OSS projects and can be easily integrated with any GitHub repositories. One-hundred one Computer Science and Computer Engineering masters students participated in our study. The survey results showed that more than 84% of students thought bots can help them to contribute code with better quality. We analyzed 396 pull requests. Results revealed that bots can provide more timely feedback than teaching staff. The Danger Bot is associated with a significant reduction system-specific guideline violations (by 39%), and the Code Climate Bot is associated with a significant 60% decrease of code smells in student contributions. However, we found that the Travis CI Bot did not help student contributions pass automated tests.**

***Index Terms*—Internet bots, open-source software, software engineering, open-source curriculum, automated feedback, Expertiza**

I. Introduction

Software engineering courses typically require students to work on programming assignments and submit code for grading. It is increasingly common for assignments to specify GitHub pull requests as deliverables. It is essential for students to receive timely feedback on their code. However, it is difficult for teaching staff to provide instant feedback, especially in large classes.

As a result, much research [1]–[5] has explored different ways to provide automated and timely feedback to students' programming assignments/projects. There are three main methods to achieve this goal: (1) the first method is to use domain-specific techniques, such as static code analysis, dynamic code analysis; (2) the second way is to work with an Intelligent Tutor System (ITS); (3) the third approach is to get help from AI techniques, such as deep learning or natural language processing [6]. The majority of previous studies have focused on introductory programming courses. Few studies have measured the usability of these tools in advanced undergraduate or graduate courses and explored approaches to create automated and timely feedback in more complex programming environments.

Each semester 50–120 students enroll in our Object-Oriented Design and Development course. There are typically four or five teaching staff, including the instructor and several teaching assistants. Each team works on a different project; hence, each semester we need to prepare 20–70 topics for OSS contributions. Each staff member mentors five or more student teams throughout the semester. Based on our previous experience, we recognize the importance of providing timely feedback for student code. Therefore, we schedule a weekly face-to-face or online meeting with each team to help students solve problems and keep them on the right track.

However, once-weekly feedback is far from optimal. We seek an approach that can automatically provide instant feedback. Since 2013, we have used a tool named Code Climate [7] as a static code analyzer. Whenever students modify and commit the code, static code analysis can detect different kinds of problems, such as code complexity, code duplication, bad code style, and security issues. We have integrated Travis CI [8], a continuous integration service, with GitHub since 2013. It can offer instant feedback by executing automated tests each time students change the code.

Although the abovementioned methods can help student modifications follow good coding style without breaking the existing test suites, they cannot provide system-specific feedback and explicitly display all detailed information on one page. Students working on OSS-based course projects need to write code compatible with an existing code base instead of starting from scratch. But, passing static analysis and automated tests cannot guarantee that a contribution is good enough to be incorporated into the OSS code base. Many other factors may affect the quality of the pull request; for example, whether it includes high-quality tests, tests for new features, or unnecessarily duplicates or modifies code not involved with the course projects. In this paper, we report how we solve the limitations of existing tools by implementing three Internet bots in the 2018 fall semester. These bots can (1) help students detect more than 40 system-specific guidelines, (2) explicitly display instant test execution results on the GitHub pull-request page, and (3) insert pull-request comments to remind students to fix issues detected by the static code analyzer. The

978-1-7281-1746-1/19/$31.00 ©2019 IEEE

survey results showed that 70% of students claimed that the advice given by the bots was useful and more than 84% of students thought bots can help them to contribute code with better quality. We analyzed 396 pull requests. Results exhibited that bots had a shorter response time than teaching staff. The Danger Bot is associated with a significant 39% decrease of system-specific guideline violations. The Code Climate Bot is associated with a significant 60% reduction of code smells. However, we noticed that the Travis CI Bot did not increase the automated test pass rate.

II. Related Work

Previous research has explored different approaches to providing automated and timely feedback to students programming assignments or projects. Keuning [6] summarize three different techniques to achieve this goal. The first one is to adopt domain-specific techniques, such as static code analysis, dynamic code analysis using automated testing, program transformations, and intention-based diagnosis. Gulwani [2] propose a programming language extension to allow instructors to define algorithmic strategies. They also introduce a novel dynamic analysis, which can automatically provide feedback on student programs by deciding whether the program meets the instructor's requirements. Results show that their dynamic analysis is accurate enough to capture the algorithmic strategy employed by the student program. More importantly, the new method requires considerably less instructor effort; the instructor only needs to go through a small portion of functionally correct student programs to provide feedback on all student programs. Singh [4] present a new feedback generation tool. Their method can find minimal changes required to correct a student program, based on a two-step translation to the SKETCH [9] synthesis language. In the first step, a student program is translated into a language called $\widetilde{\text{MPY}}$. In the second step, the $\widetilde{\text{MPY}}$ program is translated into a sketch program. In the end, the tool generates feedback of student program. Results show that the new method can correct on average 64% of incorrect student programs.

The second approach is to work with ITS techniques. Gerdes [10] introduce an intelligent tutor that can automatically generate hints and feedback for student programs. They surveyed students about their attitudes towards the intelligent tutor. On average, students gave 3.4 on five-point Likert scale questions (the higher score indicates that students are more satisfied with the intelligent tutor). The third approach is to use AI techniques, such as deep learning or natural language processing. Bhatia and Singh [11] utilized recurrent neural networks (RNNs) to model token sequences of syntactically correct programs. Then they adopted a trained RNN model to provide feedback on syntax errors. Results show that the RNN model can repair almost 32% of syntax errors in elementary programming problems.

Many researchers have built systems to provide automated and timely feedback for students' programming assignments/projects. Ihantola [12] surveyed the primary features of automatic programming-assessment tools introduced from 2006 to 2010. Since most systems are not open source, many newly created systems re-created similar functionality that already existed. Keuning [6] presented a literature review of 69 automated feedback-generation tools. They concluded that these tools do not always provide feedback on how to fix problems, and instructors cannot easily adapt these tools to meet their needs due to insufficient documentation. More importantly, most previous studies have focused on introductory programming courses. Few studies have measured the usability of these tools in advanced courses and explored ways to create automated and timely feedback in more complex programming environments. To the best of our knowledge, our study is the first work to adopt Internet bots to create automated and timely feedback on GitHub pull requests, and thus help students make better contributions to OSS projects.

III. Software Engineering Course

In our software engineering course, we require students to finish two OSS-based course projects (initial and final), most of them Expertiza-based [13] projects (other projects have been based on Mozilla, Sahana, Apache, and OpenMRS, among others) [14]. This study focuses on Expertiza-based course projects. Students work in teams of 2–4 members. Teams can bid for project topics they are interested in. Then we conduct intelligent assignment [15] to match teams with project topics. After students finish their projects, we review their work and decide whether to merge, partially merge,[1] or reject student contributions. We base our decisions on a set of 13 common mistakes [16], code analysis reports, and test execution results.

IV. Methodology

The purpose of our study is to utilize Internet bots to improve feedback (comments) on GitHub pull requests, thereby helping students to make better contributions to OSS projects. Our research questions were:

- **RQ1:** Does the Danger Bot help students comply with system-specific guidelines on contributions?
- **RQ2:** Does the Travis CI Bot help student contributions pass existing test suites?
- **RQ3:** Does the Code Climate Bot help students write code with fewer code smells?
- **RQ4:** What is the attitude of students toward the use of Internet bots to provide automated and timely feedback on their OSS contributions?

A. Internet Bots

Danger Bot: We created the Danger Bot based on a Ruby gem named *Danger*.[2] The Danger Bot was programmed to detect more than 40 system-specific problems. Examples include (1) retaining debug code in pull requests, (2) adding new features without corresponding automated tests, (3) including skipped or unimplemented tests, (4) changing the database

[1]If the entire project is not acceptable, we merge part of students' code directly into the code repository, and refactor or remove the remaining part.

[2]https://github.com/danger/danger

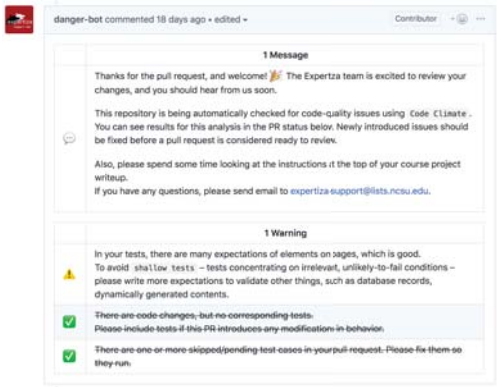


Fig. 1: A pull-request comment created by the Danger Bot

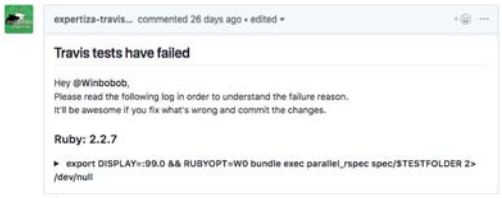


Fig. 2: A pull-request comment created by the Travis CI Bot

schema file when there are no new database migrations, (5) modifying package management files without the approval of the OSS core team. You can find the full list of Expertiza-specific guidelines in the Expertiza GitHub repository.[3] Fig. 1 shows feedback created by the Danger Bot. There are two kinds of information. The first message welcomes student contributions, presents some general information and provides a way for students to seek help. The second are warning messages that identify potential problems in student contributions. The solved warning has a strikethrough font style with a green check icon in the front. If students do not eliminate warnings, it reduces the likelihood of merging their pull request.

Travis CI Bot: We based the Travis CI Bot on an OSS project named *TravisBuddy*.[4] Our Travis CI Bot can excerpt from the long test execution log and insert information related to failed tests on pull-request pages, as shown in Fig. 2. Each comment contains all necessary information for students to debug their code. By clicking on the black triangle, students can see which test failed and the corresponding error messages. Hence they do not need to visit other pages for detailed reports.

Code Climate Bot: The Code Climate Bot can insert pull-request comments (shown in Fig. 3a) and inline comments (shown in Fig. 3b) to remind students to fix issues detected by the static code analyzer. The pull-request comment displays a summary report, including the category and count of each problem. The inline comment inserts the detailed code smell information at a specific location in the student code.

[3]https://github.com/expertiza/expertiza/blob/master/Dangerfile
[4]https://github.com/bluzi/travis-buddy

(a) A pull-request comment created by the Code Climate Bot shows the categories of problems and their corresponding counts

(b) An inline comment created by the Code Climate Bot pinpoints a problem detected by the static code analyzer

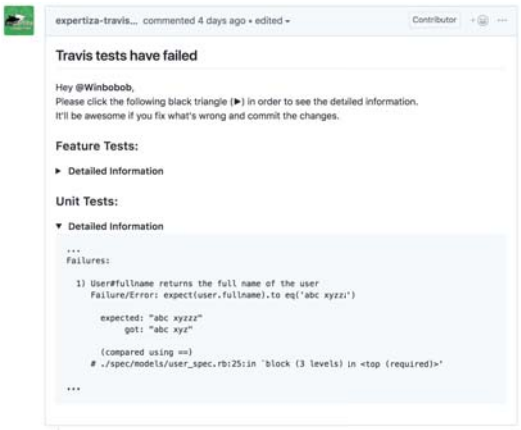
Fig. 3: Comments created by the Code Climate Bot

B. Bot Behavior Adjustment

We integrated three bots into our GitHub repository during the 2018 fall semester. We encouraged students to create pull requests as early as possible, even if they had only made minor modifications. This allowed students to get feedback from bots at the very beginning; and it let us keep monitoring the behavior of bots and making adjustments when necessary. During the initial course project, we noticed several problems with our bots. The Danger Bot created several false positive warnings and failed to detect multiple system-specific guideline violations. The Travis CI Bot had more than 18-minute response time on average. The Code Climate Bot created an overwhelming number of comments. We adjusted their behaviors accordingly before the final course project.

Improving Feedback of the Danger Bot: The Danger Bot mainly examines five aspects of the GitHub pull request: (1) pull request title, (2) the number of line additions/deletions, (3) Git commits, (4) added/modified/renamed/deleted files, (5) Git *diff* information. To implement system-specific guidelines, the Danger Bot checks to see whether the number of line additions/deletions exceeds the threshold and performs regular expression and keyword matching in different texts, including pull request titles, commit messages, file names and Git diff information. The root cause of the Danger Bot malfunction has two parts. The first part is that some regular expressions and keywords are imprecise (too big matching space), which leads to many false positive warnings. The second part is that we only considered modified files, and did not check added/renamed/deleted files. Therefore, the Danger Bot cannot detect guideline violations from these files. We then improved feedback of the Danger Bot by fixing these glitches.

Reducing the Response Time of the Travis CI Bot: There are more than 1000 automated tests in Expertiza test suite. It takes an average of 18 minutes to execute the entire test suite. We have already turned on the "fail fast" option of the



(a) A pull-request comment created by the Travis CI Bot shows that one or more tests failed. By clicking on the black triangles, developers can see which test failed and the corresponding error message.

(b) A pull-request comment created by the Travis CI Bot shows that all tests pass.

Fig. 4: Comment created by the new Travis CI Bot

RSpec [17] testing framework to stop running the test suite on the first failed test. However, since the order of executing tests is random, it is possible for the failed test to occur at the very end. In this case, students have to wait a long time before obtaining test execution results, which violates our goal—using bots to offer instant feedback.

We configured Travis CI to run different types of tests in parallel (feature tests, integration tests, unit tests, and helper tests). For instance, when a unit test fails, all subsequent unit tests are stopped by the "fail fast" option. However, the failed unit test does not prevent the execution of feature tests, integration tests, or helper tests. This optimization helps students to debug and reduces the response time of the Travis CI Bot from more than 18 minutes to approximately eight minutes. The Travis CI bot only displays detailed error information for failing tests. In Fig. 4a, for example, it shows error information for feature tests and unit tests, the two types of tests that failed. After resolving problems, students can choose to rerun only certain types of tests. If all tests pass, the Travis CI Bot congratulates students (shown in Fig. 4b). The upgraded Travis CI Bot can provide more timely feedback and help students focus on failed tests.

Eliminating Overwhelming Inline Comments Created by the Code Climate Bot: We conducted a poll on Piazza and asked students whether the comments generated by bots were overwhelming. We received votes from one-third of the students. The results indicated that almost 66% of students considered that comments given by bots are at least somewhat overwhelming.

We tallied the number of pull request comments generated by three Internet bots and by humans (teaching staff and students) separately. We found that the conversations between teaching staff and students on the GitHub pull request are very limited. Rather, most comments on pull requests come from the Code Climate Bot, since we set the bot to create inline comments whenever it found code smells. However, having too many inline comments increases the page loading time and reduces the readability of the code. Therefore, we set 20 (the average number of pull request comments in the initial course project) as the maximum number of inline comments that the Code Climate Bot can create on one pull request. When students commit new code, the Code Climate Bot can post additional inline comments on new issues. In this case, the bot can continuously detect code smells from student contributions, without giving students an overwhelming number of comments.

V. Data Collection

One-hundred one Computer Science and Computer Engineering masters students participated in this study. We collected both qualitative and quantitative data. Qualitative data came from a pre-survey and a post-survey. Table I shows all survey questions. In the pre-survey, we asked about students' work experience and their previous experience in using or developing Internet bots. The post-survey was intended to capture the students' thoughts about Internet bots from different perspectives, such as the usefulness of the advice given by the bots, and the helpfulness of the bots on contributing better quality code. We also encouraged students to mention the challenges they met when modifying code based on suggestions given by Internet bots, and to write down other concerns, suggestions, and ideas about using Internet bots in course projects.

The quantitative data came from pull requests in the Expertiza repository.[5] We studied pull requests created from the 2012 fall semester to the 2018 fall semester. There were a total of 396 pull requests, 57 of which came from the 2018 fall semester. We used the GitHub API to fetch data, including the author, timestamp, and content of each pull request comment created either by bots or by humans, from day 1 of the course projects to day 66. These days spanned the period when the vast majority of comments were written. For the fall 2018 comments, we were able to tell how many times they had been edited. This was facilitated by an enhancement to GitHub in May 2018, which allowed users to see prior revisions of a comment.[6] We decided that editing an existing comment should be treated as equivalent to creating a new comment. This means that the effective number of comments can be greater than the number reported by GitHub.

To check the effectiveness of the Danger Bot in eliminating system-specific guideline violations, we tallied the number of guideline violations in each pull request created during the

[5]https://github.com/expertiza/expertiza
[6]https://github.blog/changelog/2018-05-24-comment-edit-history/

TABLE I: Survey questions

Survey Type	Question Statement	Question Type
Pre-survey	- How long have you spent working in industry (including internships)?	Multiple choice
	- Have you used internet bots before, e.g., chat bots?	Multiple choice
	- Have you developed internet bots before, e.g., Slack bots?	Multiple choice
Post-survey	- How useful did you find the advice given by the bots used in this project?	5-point Likert scale
	- Internet bots can help you to contribute code with better quality.	5-point Likert scale
	- What challenges did you have when modifying code based on suggestions given by internet bots?	Open ended
	- How did you address those challenges, and why did you approach them the way you did? How else might you approach the problem if given those challenges again?	Open ended
	- Share with us any concerns, suggestions, ideas about using Internet bots in course projects.	Open ended

2018 fall semester. We also wrote a script to automatically analyze and tally guideline violations in semesters before 2018 fall. We excluded pull requests with more than 10,000 LoC changes because the analysis time was very long, and these pull requests typically contained large files from third parties, which could interfere with the analysis. We noticed that some students deleted forked GitHub repositories after the end of the course. We excluded those pull requests because the script could not analyze their contributions. In the end, we obtained system-specific guideline violation data from 361 pull requests (304 pull requests from semesters before 2018 fall and 57 pull requests from the 2018 fall semester).

We collected test execution results for 302 pull requests (256 from semesters before 2018 fall and 46 from the 2018 fall semester).[7] This information helped us figure out whether the Travis CI Bot was able to help students fix failed test cases by explicitly displaying detailed test execution results. There are three kinds of test execution results. The first one is *build passed*, which means Travis CI can successfully set up the test environment and run all automated tests. The second result is *build failed*, which indicates that at least one test failed. The last one is *build could not complete due to an error*. This result means that some commands fail to execute and Travis CI cannot complete the test execution process. We classified the first result as a passed test and the other two results as failed tests.

We wanted to obtain the Code Climate analysis report for each pull request created since the beginning of 2016[8] to examine the effectiveness of the Code Climate Bot. We ran into trouble because Code Climate purges analysis data after a month. To trigger a new Code Climate analysis, the pull-request *author* could add a new commit for each pull request. However, this would be difficult, because some of the authors graduated several years ago. Another way would be to clone these pull requests on our local machines and create "new" pull requests just to trigger Code Climate analysis. But this would litter our repository with duplicate pull requests, which would be confusing to future developers. So we decided to forgo the elaborated analysis report for each pull request, and instead, built a web crawler to fetch only the number of code smells detected by the Code Climate, which is available on the GitHub pull request page. In the end, we obtained the code smell data from 180 pull requests (123 pull requests from semesters before 2018 fall and 57 pull requests from the 2018 fall semester).

[7]We cannot obtain the test execution results for pull requests when they had merge conflicts.

[8]Although Code Climate has been integrated with our GitHub repository since 2013, we did not use Code Climate to analyze pull requests until 2016.

VI. Data Analysis

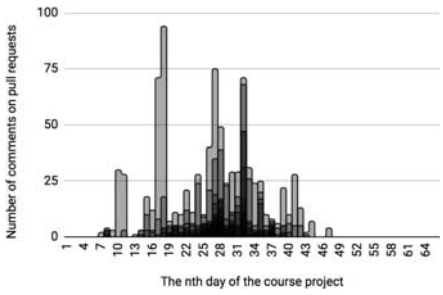
A. Survey Results

We sent a pre-survey to students at the beginning of the 2018 fall semester. The number of responses exceeded the number of students who did course projects, because some students dropped the course. We found that 60% of students had more than three months of work experience, 58% of students had used bots, and more than 8% had developed bots.

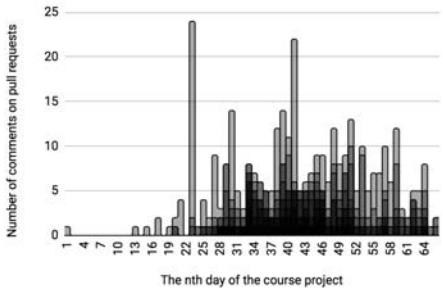
The response rate on the post-survey was almost 70%. Seventy percent of those students found the advice given by Internet bots to be useful. More than 84% of students thought that bots could help them to contribute better-quality code. One student (student 16) stated that "*the bots were useful to know whether the build failed and detected other issues.*" Student 29 mentioned that "*bots were helpful in finding code style violations.*" Students also gave many pertinent suggestions. Student 18 thought that the metrics used by the Code Climate Bot "*were extremely strict for a project with such a mixture of good and bad code.*" Student 67 said that "*thank to strictness, I learned the standard syntax a lot by fixing many errors.*" Other students claimed that some suggestions given by bots were unclear, and did not help in finding a solution. Specifically, student 16 pointed out that "*the Danger Bot does not pinpoint the location but gives the reason for failure. It would be good if it could at least point to related files.*" Student 25 suggested that "*we could have some way to run bots, like a bash script over our local repository*" to fix multiple issues locally at once and maintain a clean commit history.

B. Overview of Pull Request Comments

We tallied the comments created by bots, teaching staff, and students on each pull request from day 1 of a course project to day 66 of the same project. Fig. 5a exhibits the comments on 57 pull requests in the 2018 fall semester. Fig. 5b displays the comments for the remaining 339 pull requests from semesters before 2018 fall. In both figures, the x-axes represent the nth day of the course projects, and the y-axes show the number of comments on pull requests. Each bar represents the comments

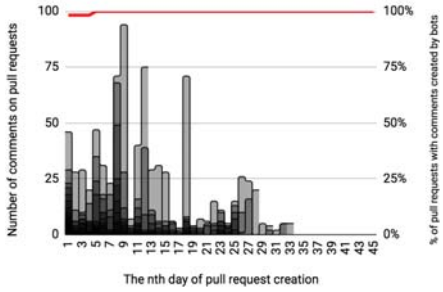


(a) Timing of comments on course projects in fall 2018

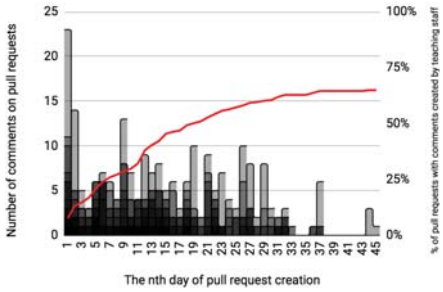


(b) Timing of comments on course projects in semesters before 2018 fall

Fig. 5: All comments for all course projects (Day 1 is the first day of the course projects.)



(a) Timing of comments by bots on course projects in fall 2018



(b) Timing of comments by teaching staff on course projects in semesters before 2018 fall

Fig. 6: Comments for all course projects (Day 1 is the first day of pull-request creation. The last comment was created on the 45th day since a pull request was created.)

created in a pull request on a particular day. To display the frequency of comments on different pull requests, we set the color of each bar 30% opacity. In this way, darker colors indicate that more comments were created on different pull requests that day.

We analyzed when the last comment was made on each pull request in the 2018 fall semester and semesters before 2018 fall. On average, the last comment was created on day 34 (median 35) in the 2018 fall semester and on day 45 (median 44) in semesters before 2018 fall. We performed tie correction and applied a one-sided Wilcoxon rank-sum test to check whether the timing of the last comment in these two sets of pull requests came from the same distribution. The results ($p = 4.5e^{-15} < 0.05$) indicates a significant difference. According to the mean and median values of the days, we can figure out that the comments in the 2018 fall semester shifted more to the left and ended earlier. In semesters before 2018 fall, students tended to create pull requests several days before deadlines, and teaching staff left comments on GitHub pull request pages during the grading or merging period. We have made several recent changes to help students make better contributions to OSS projects. First, we encouraged students to create pull requests as early as possible to allow bots to provide feedback on student contributions. Next, we scheduled a weekly meeting with all student teams to ensure they are on the right track. Moreover, core Expertiza team members have worked hard to improve the infrastructure for course projects.

C. Response Time by Internet Bots vs. Teaching Staff

As we mentioned above, one reason why the comments in the 2018 fall semester shifted to the left is that we encouraged students to create pull requests earlier that semester. To compare the response time of bots and teaching staff and eliminate the bias introduced by the pull-request creation date, we re-tallied comments in 396 pull requests by setting the start date to the pull-request creation date and the end date to day 66 of a course project. Fig. 6a and Fig. 6b show re-tallied comments generated by bots in the 2018 fall semester and re-tallied comments created by teaching staff in semesters before 2018 fall. The *x*-axes record the *n*th day of pull request creation. The *y*-axes present the number of comments on pull requests. Each bar represents, with 30% opacity, the number of comments created in a pull request on a particular day.

Based on comment data depicted in Fig. 6a and Fig. 6b, we then calculated a new metric—*the percentage of pull requests with comments*. This metric is shown as red lines in Fig. 6. We found that bots created comments on 98% of pull requests on the first day of pull-request creation and that the percentage reached 100% by day 5. Teaching staff inserted comments on only 8% of pull requests on day 1 and created comments on about one-fifth pull requests by day 5. The percentage does not reach 100 in Fig. 6b, which indicates that some pull requests (around 35%) did not receive any comments from teaching staff. Therefore, we can conclude that bots have a shorter response time and can make more timely feedback than teaching staff.

TABLE II: Contingency table for Travis CI results

	Tests passed	Tests failed	Total
Semesters before 2018 fall	149	107	256
2018 fall semester	21	25	46
Total	170	132	302

D. System-Specific Guideline Violations

We analyzed system-specific guideline violations in 361 pull requests. Three hundred and four pull requests from semesters before 2018 fall had on average 4.1 (median 4.0) guideline violations. Fifty-seven pull requests from the 2018 fall semester had 2.5 (median 2.0) guideline violations. The result of the one-sided Wilcoxon rank-sum test ($p = 1.4e^{-5} < 0.05$) indicated that there was a significant difference between the number of guideline violations in these two sets of pull requests. Based on the mean and median values of the number of guideline violations, it appears that the Danger Bot is associated with a significant reduction of system-specific guideline violations (by 39%).

We further looked into the number of guideline violations in the pull requests from the 2018 fall semester. We found that students were able to fix more than 30% of system-specific guideline violations identified by the Danger Bot.

E. Automated Test Results

We constructed a contingency table (Table II) for Travis CI results on 302 pull requests. The Travis CI pass rate was more than 58% for 256 pull requests from semesters before 2018 fall and almost 46% for 46 pull requests from the 2018 fall semester. We performed a Chi-squared test on the observed numbers of pull requests in the contingency table. The result ($p = 0.16 > 0.05$) indicated that there was no significant difference in the test pass rate for semesters before 2018 fall and the 2018 fall semester. In other words, the fact that the Travis CI Bot explicitly displayed test execution results on the pull-request pages did not make a significant difference in the test-pass rate.

F. Code Smells

We analyzed the code-smell data from 180 pull requests. There were on average 64 (median 21) code smells in 123 pull requests from semesters before 2018 fall, and 26 (median 6) code smells in 57 pull requests from the 2018 fall semester. Some pull requests contained hundreds of code smells, while others had less than 10. That is the reason for the large gaps between mean and median values. A one-sided Wilcoxon rank-sum test ($p = 0.0026 < 0.05$) showed that there was a significant difference between the number of code smells in these two sets. Moreover, the mean and median numbers of code smells given an idea of the magnitude of the effect: code smells in student submissions have decreased by around 60%. Further, we found that students were able to resolve more than 54% of code smells.

VII. Discussion

For RQ1, we concluded that the Danger Bot is associated with a significant 39% decrease of system-specific guideline violations. Some students suggested that the Danger Bot might specify the location of guideline violations. The Danger Bot could be enhanced to include this feature. We could also add some good and bad examples as part of feedback provided by the Danger Bot to help students learn correct coding guidelines. It is worth noting that we use a version control system to store the configuration parameters for the Danger Bot. This makes it easy for us to maintain the configuration and create multiple configurations to support different scenarios.

In practice, we still found some false positives in system-specific guideline violations. It is because the regular expression and keyword matching were applied to the entire Git diff information, including added/deleted LoC and adjacent code (not touched by students). If there were some guideline violations in adjacent code, the Danger Bot still considered that there were guideline violations in student contributions and reported false positive warnings. To solve this problem, we updated the Danger Bot to check only added LoC, instead of examining the entire Git diff information.

To answer RQ2, we were not able to conclude that the Travis CI Bot helped student contributions pass existing tests. The test-pass rate in the 2018 fall semester was even lower than that in semesters before 2018 fall. One potential reason is that, over time, we have added more automated tests, which increases the likelihood that at least one will fail.

We discussed the use of the Travis CI Bot in course projects with several students. Many of them claimed that they did not fully understand test execution logs provided by Travis CI. However, they still preferred to visit Travis CI pages for elaborated information on failed tests. One reason is that students do not need to understand all the information. They can debug the code as long as they can find the error messages and locate the particular file and line of code. Another reason is that the test execution logs on Travis CI pages have a black background with highlighted code. Students are used to viewing text in a terminal-like color coding environment instead of plain text provided by the bot. Besides, when Travis CI cannot complete the test process due to an error, its comment provides much less information. Nevertheless, a few students preferred looking at the feedback created by the bot for the sake of convenience. Due to student feedback, we plan to redesign the Travis CI Bot to provide more useful information and help student contributions achieve a higher test-pass rate.

For RQ3, we concluded that the Code Climate Bot is associated with a significant 60% decrease of code smells. We have mentioned three metrics: (1) **common mistakes** in student contributions to OSS projects, (2) **system-specific guidelines** identified by the Danger Bot, and (3) **code smells** detected by the Code Climate Bot in this paper. They highlight different, related, perspectives on course projects. Fig. 7 depicts the relationship between them. The left column shows the types of

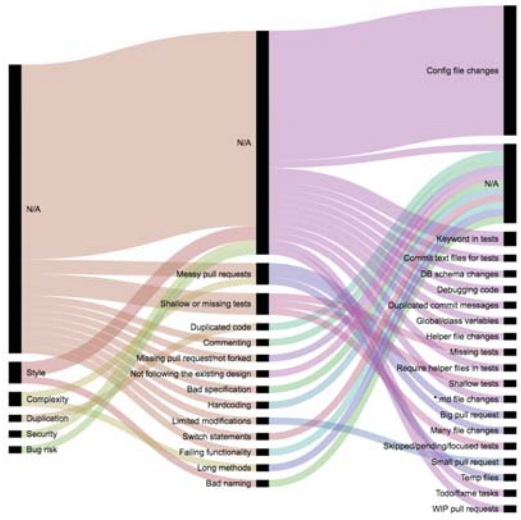


Fig. 7: Relationship between code smells (left column), common mistakes (middle), and system-specific guidelines (right). The thickness of each line corresponds to the number of the problem types.

code smells, namely, style, complexity, bug risk, duplication, and security. The middle coordinate lists 13 common mistakes. The right one shows all system-specific guideline categories. If one problem does not belong to any types, we classified it in the "N/A" category. Fig. 7 indicates that many problems do not belong to any of the five code-smell types. However, 13 common mistakes encompass around one-third of those problems, and system-specific guidelines can classify the rest. That is to say, some problems that cannot be identified by one metric can be classified by another metric. Therefore, we combined these three metrics to identify problems in pull requests, and help students make better contributions.

To answer RQ4, we believe that most students held a positive attitude toward the use of bots to give automated and timely feedback on their OSS contributions. Students fixed more than 30% of system-specific guideline violations detected by the Danger Bot, and more than 54% of code smells identified by the Code Climate Bot. Some students mentioned that bots were demanding, especially the Code Climate Bot. Other students thought that the strictness of bots could help them to learn the standard syntax. We plan to evaluate the metrics of bots and make adjustments if necessary. Students also made many good suggestions, for instance, making the Danger Bot locate the files and methods that contain issues; supporting bots to run locally instead of each time making a commit just to trigger on-the-fly analysis of bots.

VIII. Threats to Validity

Internal Validity: Concurrently with the rollout of bots, we also established a practice of meeting weekly with student teams. We are always evolving the course to teach students to avoid code smells we have noticed in projects from past semesters. These factors may cause a reduction of code smells.

External Validity: Results based on students enrolled on our course may not be generalizable to all masters students. However, we analyzed pull requests created by nearly 1000 masters students from 2012 to 2018. This fact might mitigate threats to external validity.

Construct Validity: The implementation of system-specific guidelines may not 100% match the criteria of guideline violations, which can lead to false positive warnings. Besides, several factors can affect the number of comments created by bots: (1) running tests in parallel can inflate the number of comments created by bots (We use the continuous integration service to trigger the Danger Bot analysis. Running tests in parallel can trigger the analysis multiple times and create a new comment for each analysis.); (2) pull requests with merge conflicts can reduce the number of bot comments by preventing bots from analyzing the data. Moreover, we tallied the number of guideline violations, failed tests and Code Climate issues. We assumed that more occurrences indicated lower quality pull requests. However, the raw number of issues may not be a definitive measure of quality, since a few serious issues may be more impactful than numerous small-impact issues.

IX. Future Work

In the future, we plan to upgrade the functionality of bots. We will try to make the Danger Bot locate the particular file and line of code for each problem. We plan to make bots run on local machines, like linters do. Feedback is only one-way, from Internet bots to students. Students cannot respond to the feedback, even when it contains false positives. We plan to make bots support two-way communication. When students think the comment is incorrect, they can either ask bots to double-check the code or notify teaching staff to conduct a manual inspection. Teaching staff can then decide whether the comment is accurate.

X. Conclusions

In this study, we helped students make better contributions to OSS projects by utilizing three Internet bots to provide automated and timely feedback on GitHub pull requests. These bots are either open source or free for OSS projects and can be integrated with any GitHub repositories. More than four-fifths of students claimed that bots helped them to contribute code with better quality. Results showed that bots created more timely feedback than teaching staff. The Danger Bot reduced system-specific guideline violations by almost two-fifths. Use of the Code Climate Bot led to a three-fifths decline in the number of code smells. Nevertheless, we found that the Travis CI Bot did not help student contributions pass automated test suites because many students preferred to check raw test-execution logs for detailed information. In the future, we plan to upgrade these bots to provide more useful information and establish two-way communication between bots and human beings. The assistance of bots allows teaching staff to pay more attention to design-related feedback for student contributions.

REFERENCES

[1] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: an adaptable programming tutor for haskell giving automated feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017.

[2] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 41–51.

[3] N. Falkner, R. Vivian, D. Piper, and K. Falkner, "Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units," in *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 2014, pp. 9–14.

[4] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[5] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, p. 4, 2005.

[6] H. Keuning, J. Jeuring, and B. Heeren, "Towards a systematic review of automated feedback generation for programming exercises," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2016, pp. 41–46.

[7] CodeClimate. (2019) expertiza/expertiza - code climate. [Online]. Available: https://codeclimate.com/github/expertiza/expertiza

[8] TravisCI. (2019) expertiza/expertiza - travis ci. [Online]. Available: https://travis-ci.org/expertiza/expertiza

[9] A. Solar-Lezama and R. Bodik, *Program synthesis by sketching*. Citeseer, 2008.

[10] A. Gerdes, J. Jeuring, and B. Heeren, "An interactive functional programming tutor," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 2012, pp. 250–255.

[11] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv preprint arXiv:1603.06129*, 2016.

[12] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli calling international conference on computing education research*. ACM, 2010, pp. 86–93.

[13] E. Gehringer, L. Ehresman, S. G. Conger, and P. Wagle, "Reusable learning objects through peer review: The expertiza approach," *Innovate: Journal of Online Education*, vol. 3, no. 5, p. 4, 2007.

[14] Z. Hu, E. F. Gehringer, and Y. Song, "Maintaining student-authored open-source software for more than 10 years: An experience report," in *ASEE-SE Annual Meeting*, 2019, p. To appear.

[15] S. Akbar, E. F. Gehringer, and Z. Hu, "Improving formation of student teams: a clustering approach," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 147–148.

[16] Z. Hu, Y. Song, and E. F. Gehringer, "Open-source software in class: students' common mistakes," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM, 2018, pp. 40–48.

[17] RSpec. (2019) Rspec: Behaviour driven development for ruby. [Online]. Available: http://rspec.info/