Open-Source Software in Class: Students' Common Mistakes

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

Yang Song
Department of Computer Science
University of North Carolina Wilmington
Wilmington, NC, USA
songy@uncw.edu

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

Abstract

Introducing Open Source Software (OSS) projects into a software-engineering course has many advantages, for instance, allowing students to learn good coding practices from real-world projects, and giving students a glimpse of a real project. However, it is not easy for instructors to induce one or more OSS core teams to lend support for course projects. The alternative is to have students work on "toy features"—features for these projects not specified by OSS core teams, but by teaching staff. However, the project may be unimportant to the OSS project or may disrupt its design, making those code contributions unlikely to be integrated into the OSS code repository. In this paper, we, as both teaching staff and the core team for one OSS project called Expertiza, discuss our experience in supporting 700 students on 313 OSS-based course projects in the past five years. We manually checked these course projects, and summarize 13 common mistakes that frequently occur in students' contributions, such as not following the existing design or messy pull requests. We propose five suggestions to help students reduce the frequency of common mistakes and improve the quality of their OSS pull requests.

CCS Concepts • **Information systems → Open source software**; *Collaborative and social computing systems and tools*; • **Software and its engineering → Open source model**;

Keywords Open-source software, software engineering, open-source curriculum, Expertiza

ICSE-SEET'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5660-2/18/05…$15.00
https://doi.org/10.1145/3183377.3183394

1 INTRODUCTION

Open Source Software (OSS) is computer software whose source code can be inspected, modified, and distributed by anyone under open-source licenses [13]. More and more people are starting to use open source software and making contributions to OSS projects. It has many compelling advantages. The first one is high quality. Instead of a handful of developers, hundreds or even thousands of developers are involved in the development of open-source software. With such a large number of developers, bugs can be fixed quickly, and many new features can be implemented promptly. The second advantage is the high security and stability of open-source software. The reason is similar. With permission to access every piece of source code, security holes that may have been ignored by the original authors can be patched by other developers. Another compelling advantage is that developers are free to make changes to the existing code base. In this way, developers can add any desired features, which makes the software highly customizable and suitable for a variety of purposes. What is more, an increasing number of developers are trying to make contributions to OSS projects in order to become better programmers. With the help of the support community, which is another advantage of OSS projects, developers' programming skills can be improved quickly.

As mentioned above, one significant advantage of making contributions to OSS projects is the programming skill improvement. Therefore, many researchers and educators attempt to introduce OSS projects into software-engineering courses. This effort brings many benefits to engineering students. First, reading project documents and exploring source code can help students learn a lot from real-world projects [5], such as coding style, feature design, and other good development practices. Second, working on OSS projects can provide students with a bird's eye view of a large project, since students are making contributions to an existing project instead of starting a project from scratch. They have to think about how to make the newly-added code compatible with the current design. Last but not least, students need to communicate frequently with the OSS core team. This interaction can enhance students' communication abilities, which are crucial to their future career.

Instructors who attempt to incorporate OSS projects into courses have to deal with two issues, namely, how to get ideas for OSS-based course projects and how to review students' deliverables. OSS core team members are best qualified to design course projects and review students' work. For instance, Ellis et al. [8] integrated the Sahana project into their curriculum. The Sahana core team would review project requirements and all code written by students. However, due to the voluntary nature of the open-source community, it may be difficult for other instructors to get OSS core teams to work with their course. Therefore, some instructors have specified "toy features" (features commissioned/specified by teaching staff instead of the open-source community) in OSS projects as course projects. "Toy features" render course projects "unreal," and causes students' work to be put aside once the project is finished, without any further benefit to the OSS community or users of the software.

In this paper, we discuss our experience as both teaching staff and OSS core team members in supporting students on OSS-based course projects. We reviewed 313 course projects from the 2012 fall semester to the 2017 spring semester and analyzed the reasons why we accepted (merged) or rejected students' pull requests. After that, we summarized 13 common mistakes found in reviewing students' pull requests. We further detected that the relative frequency of many common mistakes has remained stable across most semesters. Then we propose five suggestions to help students to reduce common mistakes and achieve OSS pull requests with higher quality.

2 RELATED WORK

Many papers talk about incorporating OSS projects into computer science education. Bishop et al. [3] proposed four integration options, including using open source as examples in classes, OSS-based capstone projects, extending the software for research purposes, and incorporating it into hackathons. Our literature search showed that the majority of such studies used OSS projects in regular courses [7, 9, 14, 16, 19].

Some studies discuss the benefits of integrating OSS projects into software-engineering courses. Ellis et al. [7] presented a multi-year study on involving students in an OSS project. They found that students who participated in OSS projects are able to gain significant software-engineering knowledge. Raj and Kazemian [19] used OSS projects in database-implementation courses to help students gain insights into software design and development. The authors believe the cooperation between the OSS community and academia can revitalize computer-science education. Pinto et al. [18] interviewed seven software-engineering professors who adopted OSS projects in their software-engineering courses. They figured out that inspiring students to work on OSS projects can not only improve their social and technical skills but also enhance their resume.

Other researchers focus on the challenges of adopting OSS projects in software-engineering courses. Toth [25] postulated that the first challenge of using OSS projects in software-engineering courses is to identify and select appropriate OSS projects. Smith et al. [20] reported their experience in selecting OSS projects for software engineering. They stated that it is important to select OSS projects with proper size and complexity. And the burden of selecting suitable OSS projects could impede instructors from incorporating OSS projects into software-engineering courses. Similarly, Gehringer [10] concluded that instructors need to spend weeks or months to search for proper OSS projects. And it is better for instructors to be involved in OSS development themselves. The author also offered an overview of several OSS projects which have already worked with academia. Ellis et al. [6] summarized five challenges they faced in involving students in OSS projects: student inexperience, limited course duration, informal development practices, sustaining a development effort, and product complexity.

Moreover, it is important to assess students' contributions to OSS projects. In their literature review paper, Nascimento et al. [17] summarized 10 types of assessment in both student and teacher perspectives, such as exams, reports, software artifacts, surveys, and presentations. Among them, surveys are the main instrument for getting student feedback. And software artifacts, reports, and presentations are the main artifacts assessed when grading students' submissions.

Very few papers mentioned criteria for evaluating students' contributions to OSS projects. Buchta et al. [4] introduced a detailed grading rubric involved in implementation of functionality, interaction with the project manager, the format of the report, and so on. Liu [14] proposed a software development process called GROw (Gradually Ripen Open-source Software), which includes evaluation criteria and coding-style guidelines.

To the best of our knowledge, there are no previous papers talking about common mistakes that frequently occur in students' contributions to OSS projects. In this paper, we summarize 13 common mistakes organized into six categories, which could become a checklist to help evaluate students' code.

3 EXPERTIZA AS SOFTWARE ENGINEERING CODE BASE

Initially funded by NSF, Expertiza is an open-source online peer-assessment tool [9]. It collects data for researchers to do educational data mining and data analysis, such as a reputation system to help determine the reliability of reviews [22], peer-assessment rubric improvement [21], calibrated peer assessment [24], and collusion detection in peer assessment [23]. The Ruby on Rails code base was initiated in

2007 and is available on GitHub.[1] Since Fall 2007, it has been the main source of course projects in for our masters-level Object-Oriented Design and Development course, CSC/ECE 517.

Figure 1 is a circle graph showing how Expertiza is integrated into CSC/ECE 517. Our project ideas come from reported bugs, feedback of users, rejected projects from previous semesters, and new features requested by instructors using Expertiza (which has also been used by other North Carolina State University courses and at 17 other institutions). Each core team member is responsible for elaborating detailed requirements for several project ideas in each offering of CSC/ECE 517. After that, students choose course projects and start to work on them.

Each project has a core team member to offer project support. Since all core team members are on campus, it is convenient for students to meet directly with them. We encourage students to create pull requests as early as possible. Then each time students commit code to their pull requests, several widely-used tools, such as Travis CI[2] and Code Climate[3], will be triggered automatically to run all test cases and check the code quality according to Ruby Style Guide[4]. The core team will accept qualified projects by merging them into the Expertiza code repository. After we deploy the latest code, new features will be available to all users. In this manner, a cycle is generated, which continually improves Expertiza and allows problems to be addressed in a timely manner.

Compared with other open-source software applications we have worked with (e.g., Mozilla Servo, Sahana Eden, Apache Ambari, and OpenMRS) and the Sahana project collaborated with Dr. Ellis [8], Expertiza projects have two big advantages. First, all core team members are involved in course projects and are on-campus most of the time. This allows CSC/ECE 517 students to talk to core team members face to face. Research shows that face-to-face meeting is more effective, less time-consuming and more satisfactory compared to computer-mediated communication [2]. The core team of Expertiza has four members right now, which is composed of one professor, one senior research engineer, and two Ph.D students. Moreover, there are several prerequisites to joining the Expertiza core team; for instance, a new core team member should have taken the course and ranked very high in the class, or have mentored course projects in CSC/ECE 517, or have more than one year of Ruby on Rails development experience. Second, Expertiza is used in the course as a peer-assessment tool. Each student enrolled in the course submits their homework with it and uses it to review others' work. Frequent use of Expertiza familiarizes students with the system, which is a big help when they start working on OSS-based course projects. Students' contributions can be merged into the code repository, sometimes even during the course, if they have done a great job, which is an extra encouragement for them to do their best.

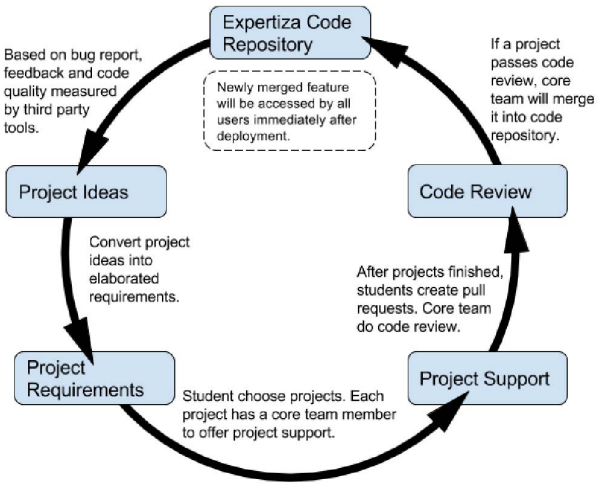

Figure 1. Expertiza development cycle.

4 COURSE SETTING AND DATA COLLECTION

Our data comes from OSS-based course projects in the past five years. Students come from different countries, but with a predominance from India, usually more than 70%. The other prominent countries of origin are the United States and China. Most of the students are majoring in Computer Science. The remaining students are mostly from Computer Engineering, Electrical Engineering, and Computer Networking. More than 90% of the students are graduate students. Most of them have bachelors degrees in Computer Science, and some of them have several years programming-related work experience.

Each semester, there are two kinds of OSS-based course projects. During the first half of the semester, the instructor will cover important points related to code refactoring. Correspondingly, the primary purpose of the initial course project is understanding the logic of existing code and refactoring it. The expected workload of these projects is to refactor around 250 lines of code (LoC) and write 100 to 200 lines of testing code. During the second half of the semester, many design patterns will be introduced in the class. And the evaluation of final course projects focuses on students' abilities to add a new feature in the system and follow the design patterns if applicable. The desired deliverable of the final course project is usually around 500 LoC changed with 100 to 200 lines of testing code.

All course projects are done in teams. The desired team size for the initial course project is 2 to 3, and for the final

[1] https://github.com/expertiza/expertiza
[2] https://travis-ci.org/expertiza/expertiza
[3] https://codeclimate.com/github/expertiza/expertiza
[4] https://github.com/bbatsov/ruby-style-guide/blob/master/README.md

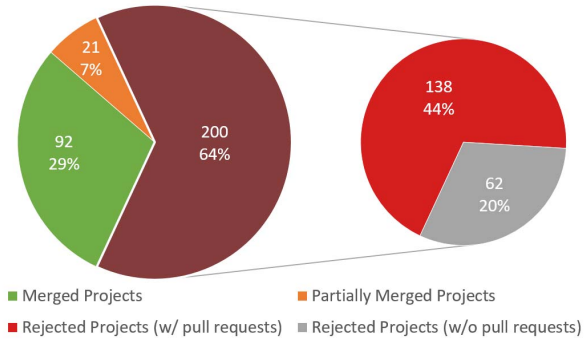

Figure 2. Merge Status of OSS-based Course Projects.

course project is 3 to 4. We ask students to form teams and select projects by themselves on Expertiza. To give each student a diversity of deveopment experiences, we required them to work with a certain number of other students during the semester. We also adopted a clustering algorithm for intelligent team formation, basd on the projects that each student expressed interest in [1].

5 COMMON MISTAKES WE OBSERVED

There were a total of 313 OSS-based course projects related to Expertiza from the 2012 fall semester to the 2017 spring semester. Among them, 92 projects were merged into the Expertiza code base (shown in green in Figure 2). Twenty-one projects were partially merged (shown in orange), which means we merged particular parts of their code directly into Expertiza repository, and refactored or removed the remaining part. Moreover, 200 projects were rejected (shown in dark red) including 62 projects that did not submit pull requests (shown in gray). We will discuss projects without pull requests later this section. The first and the second author went through the 221 rejected or partially merged projects (dark red and orange) and tallied different kinds of mistakes by referring to project grades and feedback, GitHub pull request code, and comments. In order to improve the validity of mistake coding, we discussed detected mistakes and came to agreement with each other.

We summarized 13 common mistakes from all mistakes we recorded. And we define *common* as single mistake that occurred in close to or more than 10% of rejected or partially merged projects. Our work parallels that of Gousios et al. [11], who surveyed 749 OSS core team members and summarized 23 factors that core team members check when assessing the quality of pull requests. After comparing our common mistakes and those 23 factors, we found that these two sets match well. Some differences are due to the usage of separate terminologies and different degrees of generalization. Others are the result of the difference between our education-oriented OSS project and other OSS projects. For instance, other OSS projects may consider whether contributions add value to a project when evaluating pull requests. However, all Expertiza projects are designed by core team members; they would add value to Expertiza if merged. Therefore, *value added* will not be included among the factors we consider. Later in this section, we will go through each common mistake and demonstrate corresponding code "smells" and their causes.

5.1 Common Textual Mistakes

When checking pull requests, we found that students tend to miss details because they always pay more attention to the functionality. **Bad naming** and **hardcoding** are two mistakes that occur frequently. In many cases, students made these mistakes for the sake of convenience. For throwaway projects, these mistakes may not cause many problems, however, for long-term OSS projects, these mistakes would increase maintenance costs if merged into the main code base. **Comments** frequently became an issue. Some students added several new methods without writing meaningful method comments. Other students wrote inline comments for each line of code or left commented debug code in pull requests. This breaks the continuity of the code and makes it difficult to read. Many inline comments can be avoided by adopting self-explanatory method names or variable names. We also found another situation: when students need to delete one method or file, they comment out all the code instead of removing it. It seems that students hesitate to remove code snippets from the code repository. In fact, they shouldn't need to worry about deleting code since the version-control system can retrieve it easily.

5.2 Common Coding Mistakes

An important rule of software design is the DRY principle: "Don't repeat yourself" [12]. If one feature or one piece of code is presented multiple times in one system, anyone who refractors that functionality needs to remember to make the change everywhere the functionality is implemented. Unlike throwaway projects, OSS projects can acquire **duplicated code** accidentally, when students reimplement functionality without bothering to do a thorough search to see if is already coded elsewhere in the project.

One good example in Expertiza is the functionality for reassigning topics. On some assignments, students or teams are allowed to "sign up" for a topic on which they want to work. If someone else or some other team has already chosen the topic, later choosers will be placed on a waitlist. When the topic-holder drops the topic for any reason, the first candidate on the waitlist should get this topic. The challenging part is that there are multiple scenarios in which the topic-holder may lose this topic either actively or passively: (i) when the last team member leaves the team that reserved the topic, (ii) when the student or the team switches to another topic, (iii) when the student holding the topic joins a team

that holds another topic, or (iv) when the instructor drops a team from a topic, e.g., because the team does not have enough members, and larger teams are waiting for the topic. Ideally, all these scenarios should call the same method to re-assign topics. However, this functionality was once implemented in three places in the Model-View-Controller (MVC) architecture shown in Figure 3 (once in the controller and twice in the models), which caused the topic re-assignment functionality to be difficult to maintain.

Long methods[5] are also a big issue. Most of the time, long methods contain nested loops or nested `if` statements, and handle multiple tasks instead of one. One big reason that students wrote long methods is that they tended to place a new feature in an existing method rather than to create a new one. Likewise, students prefer to modify existing `switch` or `if` blocks rather than to implement polymorphism, since it is easier for students simply to add another condition to the existing code than to understand an inheritance hierarchy structure across files. This violates a rule of object-oriented design—when you see a **switch statement**, you should think of polymorphism [15].

In many cases, we observed inconsistencies between students' demos and our manual testing results. Although the demos seemed to be fully functional, we found that some features did not work well under one or more circumstances during manual testing. Many factors will cause **failing functionality**, such as unfamiliarity with the system, and failing to implement some requirements of the project, and then bypassing them during the demo.

5.3 Common Design Mistakes

An elegantly designed system should make it reasonably easy for developers to "guess" where to find the code related to specific functionalities. When students need to invoke some functionality (e.g., to sort emails based on the last name of each user), they may not realize that this functionality or similar ones might already be implemented (e.g., in the `user` model file). In most cases, students do **not follow the existing design** and tend to rewrite the needed methods. Another reason is that even if they are able to find the method they need, it may require refactoring before using (e.g., the existing method sorts emails based on the first name of each user). Moreover, they need to make sure all other call sites for the original method work as usual. Under this circumstance, it might be easier for students to write a new method rather than refactor the existing one.

One good example is the OSS-based course project to fix the functionality of importing and exporting questionnaires. Figure 4 shows this functionality in the MVC architecture. The left side of the dotted line is the existing design in our system. In the current design, `import-file` and `export-file` controllers call corresponding methods written in different models. However, the student team did not follow the existing design and implemented the functionality in a different way (shown on the right side of the dotted line). If we merged students' pull request, this functionality would be difficult to maintain in the future. Ideally, the logical code for importing and exporting questionnaire should be placed in `questionnaire` model with correct method names and called by `import-file` and `export-file` controllers.

[5]We define a method as oversized if it exceeds 60 lines of code.

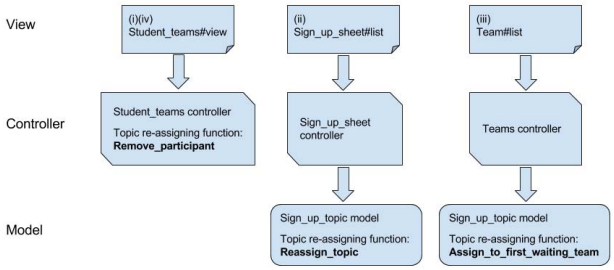

Figure 3. Functionality for reassigning topics in Expertiza.

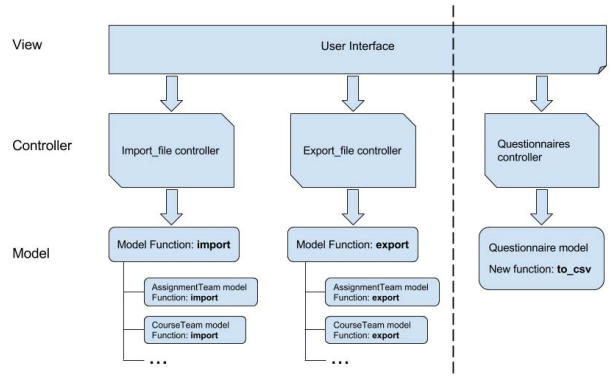

Figure 4. Functionality for importing/exporting questionnaires in Expertiza.

```
it "fills out a textbox and saves data" do
  load_questionnaire
  fill_in "response[comment]", :with => "Hello
      World!"
  select 5, :from => "response[score]"
  click_button "Submit"
  expect(page).to have_content "Response was
      successfully saved."
end
```

Figure 5. Example of Shallow Test.

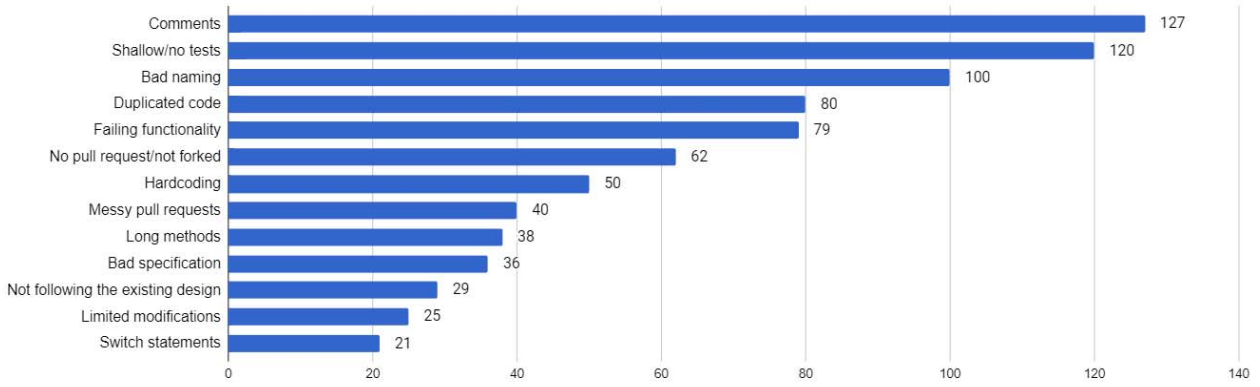

Figure 6. Frequency of Common Mistakes.

5.4 Common Testing Mistakes

Pull requests with tests have a higher chance to be merged [26]. Most of our projects ask students to write tests for their code or for previously untested code. Sometimes, even when students wrote test cases, many of them were **shallow tests**—tests concentrating on irrelevant, unlikely-to-fail conditions. If too many shallow test cases exist in the system, even if all the test cases pass, we cannot be sure that features in the system work as intended.

The OSS-based course project, "functional tests for peer assessment," is a good example. This project is supposed to test that an assessor is able to fill out the form and submit the response. And the data can be saved into the database successfully. Figure 5 shows one RSpec[6] test case students wrote. It only tested that the correct message was displayed at the top of the page after clicking "Submit" button. The test will not fail even if the data is not saved in the database, as long as the expected message appears. A more robust test of this scenario is to test not only message appearance on the view but also the corresponding database records.

5.5 Common Mergeability Mistakes

Some pull requests include unnecessary output files, additional libraries and even code modifications of other projects, which makes it difficult for core team members to figure out which code changes are directly related to the course project. **Messy pull requests** can be created when students did not realize that some files only need to be stored locally instead of being committed to the remote repository, or when they are unfamiliar with the steps of syncing a fork of the main code base. An opposite situation is when students submit pull requests with **limited modifications**. These pull requests do not improve the current code much, but only make minor changes, such as whitespace changes, or identifier name changes.

[6]http://rspec.info/

Back in 2012, we found that many students did **not create pull requests** and some of them had **not even forked** the Expertiza repository. This made it difficult for core team members to review the students' work. A more serious situation is when students downloaded the source code from GitHub, created a new repository and committed their modifications to the newly-created repository. Since the downloaded source code did not contain any history information, students were not able to create pull requests. Without the `diff` information in pull requests, the core team could not check how students' modifications affect the system, so they could not merge them. This situation occurred mostly in 2012. In later semesters, prior to assigning the projects, the instructor assigned in-class exercises on the relevant Git operations, and had the students finish them for homework. After that, this problem rarely occurred.

5.6 Common Specification Mistakes

Although each project has its requirement elaborated by core team members, some submissions deviated far from our expectation. The primary reason for **bad specification** is insufficient communication between OSS core team members and students. In recent semesters, we added a design check section before students starting coding to make sure they thoroughly understand our requirements.

6 COMMON MISTAKES ANALYSIS

We tallied the common mistakes made by each project and tried to figure out the distribution of common mistakes per semester and the relationship between common mistakes and two kinds of OSS-based course projects, namely, initial course projects and final course projects.

We counted the aggregate number of 13 common mistakes throughout 221 rejected or partially merged projects shown in Figure 6. Among all rejected or partially merged projects, almost 60% have comment issues, and more than 50% of them have testing problems. Other common mistakes like

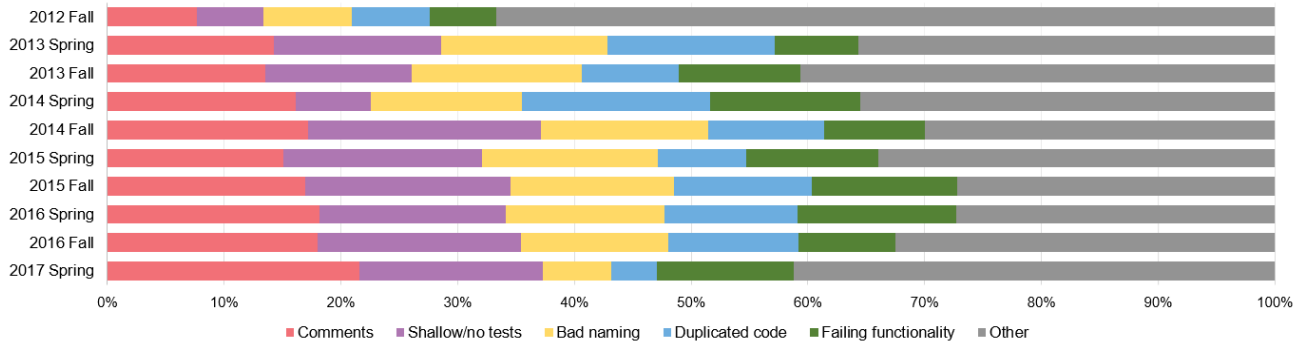

Figure 7. Relative Frequency of Common Mistakes, by Semester.

bad naming, duplicated code and failing functionality also occurred with a high frequency.

Figure 7 exhibits the relative frequency of common mistakes in each semester via percentage. We identified the five most common mistakes from Figure 6 and aggregated other common mistakes into the "Other" category. According to Figure 7, there is a big difference between the 2012 fall semester and other semesters considering back in 2012 many projects did not create pull requests. Since 2013, the proportion of five most common mistakes has remained stable across most semesters. Together they occupied more than half of all common mistakes in each semester. Therefore, it is necessary for us to devise mechanisms to help students avoid these common mistakes, especially ones with higher frequencies, and create pull requests of high quality.

We also analyzed the relationship between the number of common mistakes and two kinds of OSS-based course projects. We found that the number of common mistakes in final course projects is higher than that in initial course projects in more than half of all semesters. One reason is that final course projects make more substantial code modifications than initial ones. However, the average number of common mistakes has gone down with the introduction of third-party tools used in recent semesters.

7 OSS PULL REQUEST QUALITY CONTROL SUGGESTIONS

In this section, we propose five suggestions to help students avoid these common mistakes and create pull requests of high quality.

7.1 Elaborated Project Requirements

Based on our experience, a good project requirement statement should consist of four parts: (i) Background information, for instance, Why do we need this new feature, what is the problem with the current design, etc.? (ii) Overall requirements, explicitly telling students at a high level what they need to achieve. (iii) A list of the main files involved in this project, to show students where to start the project. (iv) Detailed requirements (displayed in the form of bulleted lists), which will directly guide students through project requirements.

7.2 Better Communication between Core Team Members and Students

The interaction between core team members and students is necessary and even crucial to the success of OSS-based course projects. The more communication students have with the project designers, the clearer idea they will have about what needs to be done. With validated design and confirmed use cases, established functionalities will have a higher chance to be merged into the code repository.

7.3 Usage of Third-party Tools

It is time-consuming to check the code style in each line of code and manually run all test cases after each code modification. Therefore, using third-party tools, such as Travis CI, Code Climate, Coveralls can help students check the code quality automatically.

7.4 Milestone Submissions

Abundant evidence indicates that students tend to start working on projects much later than they are supposed to. Milestone submissions may help students to get started earlier. We can split a big project into several milestones and let students finish them one by one. This mechanism has many advantages. First, it can offer students a clear understanding of the project requirements. Second, core team members can track the project process via milestone accomplishments and keep students on the right track. Third, during code review, we can simply check modifications in each milestone and make the final decision.

7.5 Test-driven Development

Test-driven development is one way to develop software by turning requirements into specific test cases [27]. Students

are asked to convert each task into a failed test case, make failed test case pass and refactor the code. In the end, all tasks are completed, and all tests are passed. In the meanwhile, students accomplish the high-level requirements.

8 DISCUSSION

In order to incorporate OSS projects into courses, instructors need not be core team members of an OSS project, but they should collaborate closely with one or more OSS core team members. This is because OSS core team members are best qualified to propose course project ideas, elaborate project requirements, and review students' contributions. Due to the voluntary nature of the open-source community, it is not easy for instructors to find stable contacts on OSS projects. But then again, if instructors are OSS core team members (like us), the process will be much easier.

Also, students do not need to be users of the OSS software. It is true that frequent use of the system can help students when they start working on course projects. However, knowing high-level features is not equivalent to familiarity with the low-level code base. Even if students are familiar with high-level features, they still need to understand the code before making contributions to OSS projects.

When reviewing OSS-based course projects, we discussed how to code mistakes and came to agreement with each other, so that the validity of the coding would be improved. However, it is difficult to record all kinds of mistakes exhaustively. There are several kinds of mistakes we did not cover since we are not experts in those areas, such as security. Although the Ruby on Rails framework helps protect us from many security issues, each time new code is added, new vulnerabilities may be introduced. And we will cover more kinds of mistakes in our future research.

Based on our code-review experience, some code snippets can exhibit more than one common mistake. For instance, the example we gave to demonstrate implicitly duplicated code in OSS projects (the functionality for reassigning topics) also manifests the "not following the existing design" mistake. The related code for this functionality has already been implemented in the OSS project. But the student team did not follow the existing design and duplicated that code for different scenarios. And in this case, we recorded that this course project had both two mistakes, which focus on different granularities.

Moreover, the "not following the existing design" mistake cannot be blamed on students alone. There are some other factors which can also lead to this kind of mistake, such as a shortage of development time, insufficient supporting documents, the existing bad design in OSS projects. Under these circumstances, it might be difficult for students to find existing functionalities.

In addition, we think that the 13 common mistakes we found do not exist only in OSS-based course projects. Some common mistakes can also occur in throwaway projects, such as bad naming, hardcoding. Others like "not following the existing design" may happen more frequently in OSS-based course projects. In this paper, we analyzed the data from OSS-based course projects. In the future, we could also collect data from throwaway projects, then do further analysis to check which kinds of common mistakes would happen in both OSS-based course projects and throwaway projects and which kind of mistakes would occur mostly in OSS-based course projects.

In our analysis, we gave each common mistake equal weight. In fact, some common mistakes, such as "not following the existing design" could be accorded more weight (have a bigger impact on code base) than common textual mistakes, like bad naming. It is possible for one project to have many common mistakes but have a smaller impact on the code base than another project that contains fewer, but more serious, mistakes. We could design a scoring mechanism to determine which kind of mistake has more impact on the system.

9 CONCLUSIONS

In this paper, we have presented the advantages of working on a specific OSS project in a software-engineering course. We manually checked 313 OSS-based course projects and identified 13 common mistakes, which could be made into a checklist that would help code reviewers decide whether accept or reject a particular pull request. We found that the relative frequency of many common mistakes has remained stable across several semesters. And the frequency of mistakes appears to go up during the semester, but this is because final course projects make more substantial code modifications than initial course projects.

Furthermore, we propose five suggestions based on our experience in assisting students in creating high-quality OSS pull requests. We hope these suggestions will not only help students to avoid mistakes during coding but also guide other instructors when using OSS projects as a code base for a software-engineering course.

References

[1] Shoaib Akbar, Edward Gehringer, and Zhewei Hu. 2018. Poster: Improving Formation of Student Teams: A Clustering Approach. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training Track*. ACM, To appear.

[2] Boris B Baltes, Marcus W Dickson, Michael P Sherman, Cara C Bauer, and Jacqueline S LaGanke. 2002. Computer-mediated communication and group decision making: A meta-analysis. *Organizational behavior and human decision processes* 87, 1 (2002), 156–179.

[3] Judith Bishop, Carlos Jensen, Walt Scacchi, and Arfon Smith. 2016. How to use open source software in education. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 321–322.

[4] Joseph Buchta, Maksym Petrenko, Denys Poshyvanyk, and Václav Rajlich. 2006. Teaching evolution of open-source projects in software

engineering courses. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 136–144.

[5] David Carrington and S-K Kim. 2003. Teaching software design with open source software. In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, Vol. 3. IEEE, S1C–9.

[6] HJC Ellis, RA Morelli, and GW Hislop. 2008. WIP: Challenges to educating students within the community of open source software for humanity. In *The 2008 Frontiers in Education Conference*.

[7] Heidi JC Ellis, Gregory W Hislop, Josephine Sears Rodriguez, and Ralph Morelli. 2012. Student software engineering learning via participation in humanitarian FOSS projects. In *American Society for Engineering Education*. American Society for Engineering Education.

[8] Heidi JC Ellis, Ralph A Morelli, Trishan R De Lanerolle, Jonathan Damon, and Jonathan Raye. 2007. Can humanitarian open-source software development draw new students to CS?. In *ACM SIGCSE Bulletin*, Vol. 39. ACM, 551–555.

[9] Edward Gehringer, Luke Ehresman, Susan G Conger, and Prasad Wagle. 2007. Reusable learning objects through peer review: The Expertiza approach. *Innovate: Journal of Online Education* 3, 5 (2007), 4.

[10] Edward F Gehringer. 2011. From the manager's perspective: Classroom contributions to open-source projects. In *Frontiers in Education Conference (FIE), 2011*. IEEE, F1E–1.

[11] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[12] Andrew Hunt and David Thomas. 2000. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional.

[13] Andrew M St Laurent. 2004. *Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software.* " O'Reilly Media, Inc.".

[14] Chang Liu. 2005. Enriching software engineering courses with service-learning projects and the open-source approach. In *Proceedings of the 27th international conference on Software engineering*. ACM, 613–614.

[15] Source Making. 2017. Switch Statements. (2017). Retrieved June 25, 2017 from https://sourcemaking.com/refactoring/smells/switch-statements

[16] Robert McCartney, Swapna S Gokhale, and Thérèse M Smith. 2012. Evaluating an early software engineering course with projects and tools from open source software. In *Proceedings of the ninth annual international conference on International computing education research*. ACM, 5–10.

[17] Debora MC Nascimento, Roberto Almeida Bittencourt, and Christina Chavez. 2015. Open source projects in software engineering education: a mapping study. *Computer Science Education* 25, 1 (2015), 67–114.

[18] Gustavo Pinto, Fernando Figueira Filho, Igor Steinmacher, and Marco A Gerosa. 2017. Training software engineers using open-source software: the professors' perspective. In *The 30th IEEE Conference on Software Engineering Education and Training*. 1–5.

[19] Rajendra K Raj and Fereydoun Kazemian. 2006. Using open source software in computer science courses. In *Frontiers in Education Conference, 36th Annual*. IEEE, 21–26.

[20] Therese Mary Smith, Robert McCartney, Swapna S Gokhale, and Lisa C Kaczmarczyk. 2014. Selecting open source software projects to teach software engineering. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 397–402.

[21] Yang Song, Zhewei Hu, and Edward F Gehringer. 2015. Closing the Circle: Use of Students' Responses for Peer-Assessment Rubric Improvement. In *International Conference on Web-Based Learning*. Springer, 27–36.

[22] Yang Song, Zhewei Hu, and Edward F Gehringer. 2015. Pluggable reputation systems for peer review: A web-service approach. In *Frontiers in Education Conference (FIE), 2015 IEEE*. IEEE, 1–5.

[23] Yang Song, Zhewei Hu, and Edward F Gehringer. 2017. Collusion in educational peer assessment: How much do we need to worry about it?. In *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[24] Yang Song, Zhewei Hu, Edward F Gehringer, Julia Morris, Jennifer Kidd, and Stacie Ringleb. 2016. Toward Better Training in Peer Assessment: Does Calibration Help?. In *EDM (Workshops)*.

[25] Kal Toth. 2006. Experiences with open source software engineering tools. *IEEE software* 23, 6 (2006).

[26] 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.

[27] Wikipedia. 2017. Test-driven development. (2017). Retrieved June 25, 2017 from https://en.wikipedia.org/wiki/Test-driven_development