Maintaining Student-Authored Open-Source Software for More Than 10 Years: An Experience Report

Zhewei Hu,[1] Edward F. Gehringer,[1] and Yang Song[2]
[1]North Carolina State University / [2]University of North Carolina Wilmington

Abstract

Instructors who teach software engineering courses may struggle to come up with project ideas. One approach is to use existing open-source software (OSS) projects as the code base on which to build course projects. Contributing to OSS projects holds many benefits for students. However, it is difficult for instructors to find specific open-source projects that are suitable for software engineering courses. It is also challenging for the OSS core team to specify enough course projects with a reasonable amount of work and offer consistent support during the semester. In this paper, we, as both the teaching staff and the core team of one OSS project, share our 10+-year experience on how to maintain student-authored open-source software with limited resources. We also discuss the challenges instructors may encounter when they attempt to maintain and run an application implemented by students, and what approaches they can adopt to overcome these challenges.

Keywords

Open-source software, software engineering, open-source curriculum, project-based courses, Expertiza

1 INTRODUCTION

Instructors who teach software engineering courses may have difficulty proposing ideas for course projects. This situation may happen more frequently for instructors who teach advanced undergraduate or graduate courses. This is because such courses need to focus more on object-oriented design, and design patterns compared with CS1 or CS2 courses. One approach is to use existing open-source software (OSS) projects as the code base on which to build course projects. Contributing to OSS projects holds several benefits for students. However, it is difficult for instructors to find specific open-source projects that are suitable for software engineering courses. It is also challenging for the OSS core team to specify enough course projects with a reasonable amount of work and offer consistent support during the semester.

In this paper, we, as both the teaching staff and the core team of one OSS project, share our 10+-year experience on how to maintain a student-authored open-source software with limited resources. During this decade, we have maintained a user base for our OSS application. Meanwhile, students have kept making contributions to this OSS project semester after semester. We discuss challenges that instructors may encounter when they attempt to maintain and run a software application implemented by students, from four perspectives: 1) code quality, 2) code review and deployment process, 3) infrastructure and 4) human resources. We also talk about what approaches instructors could use to handle these challenges.

© American Society for Engineering Education, 2019

2 EXPERTIZA AS A SOFTWARE ENGINEERING CODE BASE

We maintain and run a student-authored open-source software application named Expertiza,[1] whose code base is available on GitHub.[2] Expertiza is an online peer-assessment tool initially funded by NSF. The current Expertiza application was conceived in 2007. Since then, it has become the main source of course projects in our masters-level *Object-Oriented Design and Development* course, and has also been used at 21 other institutions around the world. Over the years, more than 340 students have contributed code as deliverables through GitHub pull requests and helped Expertiza undergo several major updates.

3 SOFTWARE ENGINEERING COURSE STRUCTURE

Each semester, students need to finish two OSS-based projects, most of them Expertiza-based projects (other projects have been based on Mozilla, Sahana, Apache, and OpenMRS, among others). Ideas for Expertiza-based projects come from 1) code smells as reported by static code analyzers, 2) insufficient test coverage as detected by third-party tools, 3) runtime exceptions caught by error monitoring and detection tools, 4) user feedback, 5) new features requested by users, and 6) unmerged projects from previous semesters. We prepare 20–70 course projects per semester based on the number of enrolled students. Course projects are done in teams. Teams are asked to choose which course projects they are interested in, and we run a clustering algorithm for intelligent team formation[3] to assist teams in finding a suitable project. After students finish course projects, the instructor, as well as other teaching staff review their deliverables. Most of the time, we finish the review process within one week of the deadline, to provide timely feedback to students. If students have done a good job, we merge their contributions into the code repository. If the entire project is not acceptable, we partially merge their contributions. This means that we merge part of student code directly into the code base and refactor or remove the remaining part. In the worst situation, we reject student contributions. We usually merge projects whose score is ≥ 93/100. In the other two situations, the grade will normally be lower than 93.

4 MAINTAINING STUDENT-AUTHORED OSS PROJECTS

In this section, we discuss what **challenges** instructors would meet when they attempt to maintain and run a student-authored software application and what **approaches** instructors should use to overcome these challenges. They are closely related and affect each other. If we do not meet these challenges, the system may have poor usability. We discuss 1) code quality, 2) the code review and deployment process, 3) infrastructure and 4) human resources these four perspectives. Figure 1 shows the relationship among these four perspectives. Different background colors represent different perspectives. Each arrow indicates that the challenge at the starting point of the arrow may trigger the challenge shown at the end of the arrow. For instance, if we cannot handle environment upgrades well, some common defects can be introduced into the system. Then common mistakes can trigger some runtime exceptions in the production environment. In the end, too many runtime exceptions result in poor system usability. Table 1 summarizes challenges and corresponding approaches in each perspective. We discuss them in detail below.

4.1 Code Quality

The first challenge instructors face is how to maintain a high-quality code base. If the software

© American Society for Engineering Education, 2019

Table 1. Challenges and Approaches in Each Perspective.

Perspectives	Challenges	Approaches
Code Quality	● Low pull request merge rate ● Common mistakes ● Environment upgrades	● Plug-and-play setup for OSS environment ● Static code analysis ● Internet bots
	● Runtime exceptions	● Error monitoring tool
Code review and deployment process	● Insufficient test coverage	● More high-quality automated tests ● Avoid manual tests on local machines
	● Whether and when to deploy new features	● Canary release ● Blue-green deployment
Infrastructure	● Single point of failure	● A second server ● Regular data backup ● Disaster Recovery Plan (DRP)
Human resources	● Small team ● Student developers ● High mobility	● Attract more Ph.D. or undergraduate students ● More automation

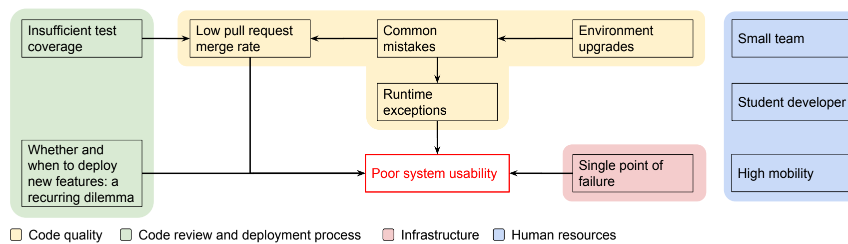

Figure 1. Relationship among Different Perspectives.

application is based primarily on student code developed over many years, it is inevitable that there are some code quality issues (code smells) in the code base. We discuss the concrete manifestation of code smells and the problems they cause. We also mention several methods we have adopted to overcome these challenges.

According to one report, on average 92% of pull requests of industrial projects are merged into the master code base.[4] However, pull requests submitted by students have a **low merge rate**. We were only merging around 30% of course projects into the Expertiza code base.[5] The report mentions several possible reasons for pull requests not being merged, including unclear development directions and late-changing requirements. These causes are not applicable in most educational settings, because all course projects are designed by teaching staff who should have a clear idea of what needs to be developed. Most of the time, we will not change project requirements in mid-stream.

The main reason for the low merge rate is that students do not make high-quality contributions. In our previous research,[5] we manually checked 313 course projects from the past five years and summarized 13 **common mistakes** that frequently occur in students' deliverables. The proportion of the five most frequent mistakes—which include commenting, shallow/no tests, bad naming, duplicated code, and failing functionality—exceeds 50% in each semester. Hence, we need some tactics to help students eliminate these common mistakes and make better contributions. Also, a domino effect may occur if we merge too few contributions: the release of new features may be delayed, causing the software application to lose users.

We have used Git to manage the Expertiza code base for more than 10 years. During this decade, the Ruby on Rails community has repeatedly upgraded the web application framework. Due to

© American Society for Engineering Education, 2019

environment upgrades, Expertiza has also undergone several major updates: from Rails 1.0 to 3.0, and from 3.0 to 4.0. There is a certain amount of source code still using the old syntax. Code with old syntax either works well but with some deprecation warnings, or breaks some features. These are challenges in maintaining the old code base.

To support students' participation in OSS-based course projects, we have prepared a **plug-and-play setup for the OSS environment**, using virtual machines (VirtualBox) and Docker images. The idea is that these can provide a self-contained development environment for the projects. These environments help students to start projects smoothly.

We also make use of **static code analysis**. With its help, we are able to enforce many coding guidelines to the code repository. Since 2013, we have used a tool called Code Climate,[6] which is free for OSS projects. The tool can perform static analysis and detect problems such as code complexity, code duplication, bad code style, and security issues. Although we have set up a static code analyzer to help check students' contributions, it cannot enforce system-specific guidelines. Therefore, we implemented three **Internet bots** during the 2018 fall semester to bypass the limitations of existing tools. Our pilot study[7] results show that 1) more than 70% of students think the feedback given by the bots is useful; 2) bots can provide six times more feedback on average than teaching staff; 3) bots can help student contributions avoid more than 33% of system-specific guideline violations.

In the production environment, many **runtime exceptions** may occur every day. We have used an **error monitoring tool** named Airbrake[8] since 2011. Its free plan includes a quota of 5000 monthly errors and two-day error retention. Airbrake sends instant alerts (emails) with stack traces and other information to the OSS core team whenever a runtime exception occurs. We take advantage of the information provided by Airbrake to eliminate these runtime exceptions. However, Airbrake's free plan lacks many advanced features, such as a detailed summary report, deployment tracking, and third-party tool integration. And it will no longer send alerts once the number of errors exceeds the monthly quota. Alternatively, we could extend the capabilities of existing open-source error monitoring tools to add advanced features and handle unlimited runtime exceptions.

4.2 Code Review and Deployment Process

Our typical code review and deployment pipeline consists of six steps as shown in Figure 2: 1) creating/modifying a pull request; 2) automatically executing static code analysis and automated tests after each code commit; 3) performing manual testing; 4) optionally discussing the code in the weekly meeting of the OSS core team; 5) deploying the code. If we find problems during steps 2–4, we will ask the author to modify the code and go through these steps again (shown by the dotted line). Then we deploy the code to the production environment. If some runtime exceptions occur because of our newly-deployed code, we can 6) roll back to the previous release.

Our current code review and deployment pipeline relies heavily on continuous integration, manual testing, and continuous deployment. The biggest problem with continuous integration is insufficient test coverage. The biggest problem with manual testing is that most of the time we test new features on local machines. And the biggest problem with continuous deployment is that we need a mechanism to decide whether and when to deploy new features.

© American Society for Engineering Education, 2019

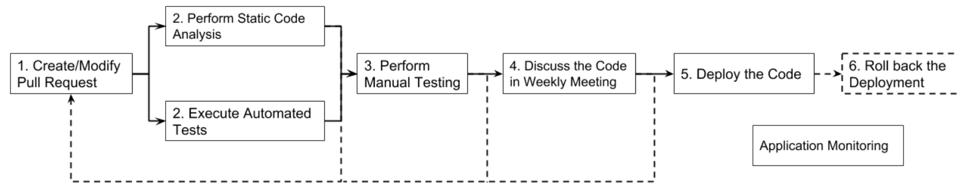

Figure 2. Code Review and Deployment Pipeline.

Expertiza has **insufficient test coverage**, only around 50%, which means the automated tests in Expertiza lacks thoroughness. Moreover, many existing tests are shallow tests—tests focusing on irrelevant, unlikely-to-fail conditions,[5] which weakens their fault-finding capability. Before merging each project, we make sure that all existing tests pass and perform additional manual tests. But these may not cover all edge cases, even fatal errors.

More high-quality automated tests are crucial to the quality control of the code base. We have used a tool called Coveralls[9] since 2014. It can visualize statement coverage statistics in different granularities, from repository overview coverage, individual file coverage to line-by-line coverage. Each time developers modify the code base, Coveralls calculates the new test coverage and sends a notification to those developers.

Furthermore, we should **avoid manual tests on local machines** to eliminate the "it works on my machine" problem. Every machine has different environment settings. If a new feature works on one developer's local machine, there is no guarantee that the new feature also works in the production environment. The solution is to use on-demand test servers. Before manual testing, we use Ansible,[10] a configuration-management tool, to spin up a test server with the same settings as the production environment and pull a specific version of the code base from the version-control system. Then multiple developers can log into the system and conduct manual testing together. During manual testing, we always conduct smoke testing first to make sure new contributions do not break the most important and most basic features of the system.

As the core team of Expertiza, we need to decide **whether to deploy new features**, which is a recurring dilemma. Suppose we have a new feature which is urgently required but not thoroughly tested. If we deploy the new feature, some runtime exceptions may occur. On the other hand, if we do not deploy the new feature, users will not benefit from it. Another example is whether we need to deploy a usable feature with dirty code. If we deploy the feature, users will benefit in the short term. Later we can ask another team to refactor the code. On the other hand, not deploying the new feature allows us to avoid code smells and the technical debt caused by the dirty code. More importantly, we also need to decide **when to deploy new features**. If we deploy new features immediately after we confirm the modifications, the change may come while the system is heavily used. If we have overlooked some edge cases, users may encounter bugs. Alternatively, we can deploy new features only when the system is not heavily used, such as late nights or early mornings. However, this requires developers to work late at night or get up early in the morning.

Since 2011, we have used a tool named Capistrano[11] to handle continuous deployment. However, the tool cannot help us to decide whether and when to deploy new features. One technique called **canary release** can help us to decide whether to deploy a new feature. Canary release can reduce the risk of introducing new features into the production environment by first rolling out the

© American Society for Engineering Education, 2019

feature to the core team, then to a small set of users, and finally to the entire user group. If developers need to address any issues during this process, the new feature will be unavailable to the next subset of users. We can deploy new features frequently by introducing the **blue-green deployment**. It uses two environments (blue and green). At any time, only one environment is live (e.g., blue). Then we can deploy new features to the other environment (green). After the green environment is stable, we can switch all incoming traffic to the green environment and the blue one becomes idle.[12] If developers encounter some runtime errors in the green environment, they can easily switch back to the blue environment.

4.3 Infrastructure

Currently, we have deployed Expertiza on a server that physically located in our university. The single server becomes a **single point of failure**. Users cannot access Expertiza whenever the server is down or scheduled for maintenance. To eliminate the single point of failure, we have already configured **a second server** to achieve higher site reliability. And we created a load balancer to control traffic between two servers. Furthermore, we have set up **regular data backup** to avoid the data loss and plan to document a **Disaster Recovery Plan (DRP)** to protect the entire infrastructure in the disaster.

4.4 Human Resources

Expertiza core team is a **small team**. Last year, the core team of Expertiza had four members. Two core team members left recently because of graduation and lack of continuing funding. One student joined the team. Currently, there are three members in Expertiza core team. In the event of an emergency, at least one of us has to take the action to resolve the problem. Most core team members are **student developers**. Although most of us have taken related courses and had several internship experiences, our experience is still limited compared with practitioners in industry. Furthermore, as students, we have to take other courses. Hence, we are unable to maintain Expertiza full time. Besides the core team of Expertiza, we have several masters students who help us refactor code and fix bugs, which is quite helpful. However, masters students have **high mobility**— most masters students stay on the team only a semester or two. It might be more effective to **attract more Ph.D. or undergraduate students** to the team since they are able to stay longer than masters students. We can always introduce **more automation** to make up for the lack of human resources. We have already implemented automatic static code analysis, automated testing framework, and automated test coverage calculation to ensure the quality of student contributions.

5 CONCLUSIONS

We have shared our 10+-year experience on how to maintain student-authored open-source software with limited resources, and how to organically make an OSS project as the code base of a software engineering course. We have also discussed what challenges instructors would face when attempting to maintain and run a software application implemented by students, from four perspectives: 1) code quality, 2) code review and deployment process, 3) infrastructure and 4) human resources. Finally, we discuss what approaches instructors could use to overcome these challenges for each perspective.

© American Society for Engineering Education, 2019

References

1 E. Gehringer, L. Ehresman, S. G. Conger and P. Wagle, "Reusable learning objects through peer review: The Expertiza approach," Innovate: Journal of Online Education, Vols. 3, no. 5, p. 4, 2007.
2 Expertiza. Available: https://github.com/expertiza/expertiza.
3 S. Akbar, E. Gehringer and Z. Hu, "Improving formation of student teams: a clustering approach," in Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, 2018.
4 "8% of pull requests are doomed,". Available: https://codeclimate.com/blog/abandoned-pull-requests/.
5 Z. Hu, Y. Song and E. Gehringer, "Open-source software in class: students' common mistakes," in Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, 2018.
6 "Code Climate,". Available: https://codeclimate.com/github/expertiza/expertiza.
7 Z. Hu and E. Gehringer, "Use Bots to Improve GitHub Pull-Request Feedback," To appear in Proceedings of the 50th ACM Technical Symposium on Computer Science Education, 2019.
8 "Airbrake,". Available: https://airbrake.io/.
9 "Coveralls,". Available: https://coveralls.io/github/expertiza/expertiza?branch=master.
10 Ansible. Available: https://www.ansible.com/.
11 Capistrano. Available: https://capistranorb.com/.
12 M. Fowler, "BlueGreenDeployment,". Available: https://martinfowler.com/bliki/BlueGreenDeployment.html.

Zhewei Hu

Zhewei Hu is a Ph.D. candidate in the Department of Computer Science, North Carolina State University. Zhewei's research interests include computer science education and software engineering.

Edward F. Gehringer

Dr. Edward Gehringer is a Professor in the Department of Computer Science, North Carolina State University. Dr. Gehringer leads the Expertiza project, which is supported by the National Science Foundation. His research focuses on computer-supported education, educational technology, computer architecture, natural-language processing, and ethics in computing.

Yang Song

Dr. Yang Song is an Assistant Professor in the Department of Computer Science, University of North Carolina Wilmington. Dr. Song received his Ph.D. degree from NC State University. His research focuses on machine learning, data mining, data science and their applications on higher education.

© American Society for Engineering Education, 2019