

## ABSTRACT

HU, ZHEWEI. Helping Students Make Better Contributions to Open-Source Software Projects. (Under the direction of Dr. Edward Gehringer).

Instructors who teach software engineering courses may struggle to come up with course project ideas. This situation may occur more frequently for instructors who teach advanced undergraduate or graduate 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 many benefits for students, for instance, allowing them to learn good coding practices from real-world projects, and giving them a glimpse of a real project. 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.

We, as both the teaching staff and the core team of one OSS project named Expertiza, have long-term experience in maintaining student-authored open-source software and making it as the code base of software engineering courses. Our course projects require students to use GitHub pull requests as deliverables to make contributions to OSS projects. One of the big challenge we face is to improve the code quality of student contributions. After manually checking 313 OSS-based course projects in the past five years, we summarized 13 common mistakes that frequently occur in student contributions, such as not following the existing design or messy pull requests. We propose four suggestions, including (1) elaborated project requirements, (2) better communication between core team members and students, (3) test-driven development, and (4) usage of third-party tools, to help students reduce the frequency of common mistakes and improve the quality of their contributions.

We have already adopted the first two suggestions over the years. To test the performance of the test-driven approach on OSS-based course projects, we conducted a quasi-experimental controlled study, which lasted for more than one month. Thirty-five masters students participated in our study. They worked on course projects in teams, half of which were assigned to the TDD group (using the test-driven approach), and the rest of which were assigned to the non-TDD group (using the traditional test-last approach). We found that students in the TDD group were able to apply test-driven techniques pragmatically—spending more than 20% of their time on average complying with the test-driven process—throughout the whole project. There were no major differences in the quality of source-code modifications and newly added tests between the TDD group and the non-TDD group; however, the TDD group wrote more tests and achieved significantly higher (12% more) statement coverage.

Besides, we have also set up a static code analyzer (Code Climate) and a continuous integration service (Travis CI) on GitHub to help students check different aspects of the code, such as code style,

security issues, and functionality. However, these third-party tools have some limitations: (1) they cannot enforce system-specific guidelines; (2) they do not make it obvious where to find detailed information. We discuss how we bypass the limitations of existing tools by implementing three Internet bots to (1) help detect violations of more than 40 system-specific guidelines, (2) explicitly display instant test execution results on the GitHub pull-request page, and (3) insert feedback to remind students to fix issues detected by the static code analyzer. 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 decrease of 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.

© Copyright 2019 by Zhewei Hu

All Rights Reserved

Helping Students Make Better Contributions to Open-Source Software Projects

by  
Zhewei Hu

A dissertation submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the Degree of  
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

---

Dr. Christopher Parnin

---

Dr. Sarah Heckman

---

Dr. Ranga Vatsavai

---

Dr. Edward Gehringer  
Chair of Advisory Committee

## **DEDICATION**

To all my family members.

## **BIOGRAPHY**

Zhewei Hu was born on March 3, 1992, in Cixi, Zhejiang, China. He attended the Zhejiang Normal University from 2010 to 2014, graduating with a bachelor's degree in Educational Technology in 2014. He joined the graduate program in Department of Computer Science at North Carolina State University in Fall 2014. He has been working with Dr. Gehringer since November 2014. He obtained the degree of Master of Science in 2016. After that, he kept working with Dr. Gehringer to pursue a Ph.D. degree.

He is the chief contributor and maintainer of the open-source peer review system—Expertiza. He contributed more than 1000 commits, refactored and merged more than 100 pull requests. In 2016 summer, he interned as a Software Developer at Offers.com at Austin, Texas. In 2017 summer, he interned as a DevOps Engineer at IBM Aspera at Emeryville, California. In 2018 summer, he interned as a Site Reliability Engineer at Carpinteria, California. He will work as a Site Reliability Engineer at Pinterest at San Francisco, California in August 2019.

With the defense of this thesis, he is receiving the degree of Doctor of Philosophy in Computer Science from North Carolina State University. His research focuses on computer science education and software engineering.

## ACKNOWLEDGEMENTS

I would like to extend my thanks to the following people without whom it would not have been possible for me to achieve this work.

I sincerely thank all my family members, especially my parents and Tiange Hai, for their dedication towards my future and their tireless efforts to expose me to the best of opportunities. I thank them for all the sacrifices they have made to provide me with everything I have ever needed. Without your help, I cannot complete this work.

I owe my sincere thanks to Dr. Edward Gehringer, my advisor, for his all-time support and guidance. I am very grateful to him for his valuable time, continuous advice and patient feedback. I learned a lot from our weekly meetings and his insights that helped me to generate creative ideas in my mind.

I would like to thank my other committee members, Dr. Christopher Parnin, Dr. Sarah Heckman, and Dr. Ranga Vatsavai, for being my committee and providing valuable feedback about the work.

I thank Yang Song, my good teacher, and helpful friend, for recommending me to join Dr. Gehringer's research team, guiding and cooperating with me to finish many papers, and going fishing with me.

Finally, I thank all other friends for their encouragement and supports at every moment.

## TABLE OF CONTENTS

<b>LIST OF TABLES</b> .....	<b>viii</b>
<b>LIST OF FIGURES</b> .....	<b>ix</b>
<b>Chapter 1 Research Background</b> .....	<b>1</b>
1.1 Open-Source Software .....	1
1.2 Open-Source Software in Education .....	2
1.3 Project-Based Programming Assignments .....	2
1.4 Problems .....	3
1.4.1 How to Get and Incorporate OSS Projects into Software-Engineering Courses? .....	3
1.4.2 How to Improve Student Contributions to OSS Projects? .....	3
1.5 Expertiza .....	4
1.6 Course Settings .....	8
1.7 Overview of the Approach .....	9
<b>Chapter 2 Long-Term Maintenance of Student-Authored Open-Source Software</b> .....	<b>10</b>
2.1 Introduction .....	10
2.2 Maintaining Student-Authored OSS Projects .....	10
2.2.1 Code Quality .....	12
2.2.2 Code Review and Deployment Process .....	13
2.2.3 Infrastructure .....	15
2.2.4 Human Resources .....	15
2.3 Conclusion .....	16
<b>Chapter 3 Common Mistakes in Student Contributions to Open-Source Software Projects</b> .....	<b>17</b>
3.1 Introduction .....	17
3.2 Related Work .....	18
3.3 Expertiza and Other OSS Projects .....	19
3.4 Common Mistakes We Observed .....	19
3.4.1 Common Textual Mistakes .....	20
3.4.2 Common Coding Mistakes .....	21
3.4.3 Common Design Mistakes .....	22
3.4.4 Common Testing Mistakes .....	24
3.4.5 Common Mergeability Mistakes .....	24
3.4.6 Common Specification Mistakes .....	25
3.5 Analysis of Common Mistakes .....	25
3.5.1 Overview of Common Mistakes .....	27
3.5.2 Common Mistakes in Rejected vs. Partially Merged Projects .....	29
3.5.3 Common Mistakes in Initial vs. Final Projects .....	29
3.6 Quality Control Suggestions for OSS Pull Requests .....	31
3.6.1 Well-designed Project Tasks .....	32
3.6.2 Better Communication between Core Team Members and Students .....	32
3.6.3 Test-Driven Development .....	33



3.6.4	Usage of Third-Party Tools . . . . .	33
3.7	Discussion . . . . .	33
3.8	Threats to Validity . . . . .	34
3.8.1	Internal Validity . . . . .	34
3.8.2	External Validity . . . . .	34
3.8.3	Construct Validity . . . . .	35
3.9	Conclusions . . . . .	35
 <b>Chapter 4 A Test-Driven Approach to Improving Student Contributions to Open-Source Software Projects . . . . . 36</b>		
4.1	Introduction . . . . .	36
4.2	Related Work . . . . .	37
4.2.1	Introducing the Test-Driven Approach to Computer Science Education . . . .	37
4.2.2	Measurement of Compliance with the TDD Process . . . . .	38
4.2.3	Challenges of Adherence to the TDD Process . . . . .	39
4.3	Methodology . . . . .	40
4.3.1	Tactics for Helping Students Complete Course Projects . . . . .	40
4.3.2	Experiment Design . . . . .	43
4.3.3	Examination of Compliance with the TDD Process . . . . .	44
4.4	Data Collection . . . . .	45
4.5	Data Analysis . . . . .	46
4.5.1	Survey Results . . . . .	46
4.5.2	Compliance with the TDD Process . . . . .	48
4.5.3	Quality Analysis of Source-Code Changes . . . . .	48
4.5.4	Test Quality Analysis . . . . .	51
4.6	Discussion . . . . .	53
4.7	Threats to Validity . . . . .	55
4.7.1	Internal Validity . . . . .	55
4.7.2	External Validity . . . . .	55
4.7.3	Construct Validity . . . . .	55
4.8	Conclusions . . . . .	56
 <b>Chapter 5 An Internet Bots Approach to Improving Feedback on GitHub Pull Requests . . 57</b>		
5.1	Introduction . . . . .	57
5.2	Related Work . . . . .	58
5.3	Methodology . . . . .	60
5.3.1	Internet Bots . . . . .	60
5.3.2	Bot Behavior Adjustment . . . . .	63
5.4	Data Collection . . . . .	67
5.5	Data Analysis . . . . .	68
5.5.1	Survey Results . . . . .	68
5.5.2	Overview of Pull Request Comments . . . . .	69
5.5.3	Response Time by Internet Bots vs. Teaching Staff . . . . .	69
5.5.4	System-Specific Guideline Violations . . . . .	72
5.5.5	Automated Test Results . . . . .	73

5.5.6	Code Smells . . . . .	73
5.6	Discussion . . . . .	73
5.7	Threats to Validity . . . . .	76
5.7.1	Internal Validity . . . . .	76
5.7.2	External Validity . . . . .	77
5.7.3	Construct Validity . . . . .	77
5.8	Conclusions . . . . .	77
<b>Chapter 6</b>	<b>Summary and Future Work . . . . .</b>	<b>78</b>
6.1	Summary . . . . .	78
6.2	Future Work . . . . .	79
<b>BIBLIOGRAPHY</b>	<b>. . . . .</b>	<b>81</b>

## LIST OF TABLES

Table 2.1	Challenges and approaches in each perspective . . . . .	11
Table 3.1	Criteria for common mistakes . . . . .	26
Table 3.2	Number of students enrolled in each spring (S) and fall (F) semester . . . . .	29
Table 3.3	Chi-square tests for common mistakes in rejected vs. partially merged projects	30
Table 3.4	Chi-square tests for common mistakes in initial vs. final projects . . . . .	31
Table 4.1	Twelve course projects . . . . .	43
Table 4.2	Patterns followed in developing code and tests . . . . .	45
Table 4.3	Post-survey questions . . . . .	46
Table 4.4	Merge status and common mistakes of 12 course projects . . . . .	48
Table 4.5	Overview of pre-existing (p), staff-authored (t) and student-authored tests (s) .	52
Table 4.6	Average percentage of commits per team per week . . . . .	54
Table 5.1	System-specific guidelines . . . . .	61
Table 5.2	Survey questions . . . . .	65
Table 5.3	Contingency table for Travis CI results . . . . .	72

## LIST OF FIGURES

Figure 1.1	Expertiza development cycle . . . . .	4
Figure 1.2	Screenshots of content management pages . . . . .	6
Figure 1.3	Screenshots of review and score reports . . . . .	7
Figure 2.1	Relationship among different perspectives . . . . .	11
Figure 2.2	Code review and deployment pipeline . . . . .	14
Figure 3.1	Merge status of OSS-based course projects from the 2012 fall semester to the 2017 spring semester . . . . .	20
Figure 3.2	Functionality for reassigning topics in Expertiza. The left flow chart shows the functionality for scenario (i) and (iv). The middle chart displays the functionality for scenario (ii). The right one presents the functionality for scenario (iii). . . . .	22
Figure 3.3	Functionality for importing/exporting questionnaires in Expertiza . . . . .	23
Figure 3.4	An example of shallow test . . . . .	24
Figure 3.5	Number of course projects that has each common mistake . . . . .	27
Figure 3.6	Number of course projects that have each common mistake, by semester. Due to space limitations, we abbreviate each common mistake: CMT (commenting), SMT (shallow or missing tests), BN (bad naming), DC (duplicated code), FF (failing functionality), MPR (messy pull request), HC (hardcoding), NFD (not following the existing design), NoPR (missing pull request/not forked), LMe (long methods), Spec (bad specification), SS (switch statements), and LMo (limited modifications). . . . .	28
Figure 3.7	Frequency of occurrence of most-common mistakes, by semester . . . . .	28
Figure 3.8	An example of well-designed project tasks . . . . .	32
Figure 4.1	An example of the method that needs to be tested (a), corresponding test skeleton (b) and completed tests (c) . . . . .	42
Figure 4.2	Data-processing pipeline . . . . .	44
Figure 4.3	An overview of student development steps . . . . .	47
Figure 4.4	A piece of source code (a) and code after modified by students (b), the corresponding test written by students (c) and code refactored by course staff (d) . . . . .	50
Figure 5.1	A pull-request comment created by the Danger Bot . . . . .	62
Figure 5.2	A pull-request comment created by the Travis CI Bot . . . . .	62
Figure 5.3	Comments created by the Code Climate Bot . . . . .	64
Figure 5.4	Comment created by the new Travis CI Bot . . . . .	66
Figure 5.5	All comments for all course projects (Day 1 is the first day of the course projects.) . . . . .	70
Figure 5.6	Comments for all course projects (Day 1 is the first day of pull-request creation. The last comment was created on day 66 of one course project, which is the 45th day since that pull request was created.) . . . . .	71

Figure 5.7	A pull-request comment created by the Travis CI Bot shows that test execution cannot be completed due to an error. . . . .	74
Figure 5.8	Relationship between code smells (left column), common mistakes (middle), and system-specific guideline categories (right). The thickness of each line corresponds to the number of the problem types. . . . .	75

## CHAPTER

# 1

# RESEARCH BACKGROUND

## 1.1 Open-Source Software

Open Source Software (OSS) is computer software whose source code can be inspected, modified, and distributed by anyone under open-source licenses [Lau04].

More and more people are starting to use open source software and make 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.

## 1.2 Open-Source Software in Education

As mentioned above, one significant advantage of making contributions to OSS projects is improving programming skill. Therefore, many researchers and educators have introduced 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 [CK03], 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.

## 1.3 Project-Based Programming Assignments

Project-based programming assignments are common in software-engineering courses. More and more instructors are experimenting with long-lasting student-authored projects, industrial projects, and OSS projects, to help students gain experience in the real-world software development process.

Heckman et al. [Hec18] shared their experience in incorporating a student-authored software application named iTrust into a junior-level software-engineering course. They discussed some benefits and challenges in making a 10+-year software as the code base of the course. Then they talked about some approaches to overcome these obstacles. Ellis et al. [Ell15] discussed their experience in involving students in real-world projects—Humanitarian Free and Open Source Software (HFOSS) projects. Based on a six-year study, the authors found that working on HFOSS projects can not only help students gain experience in developing software in a distributed environment but also increase their interest in computer science. Fox & Patterson [FP12] talked about the benefits of taking their project-based online courses—Agile Development Using Ruby on Rails, onto edX.<sup>1</sup> Their project-based courses combine techniques like Agile development, the Ruby on Rails web framework, software as a service, and cloud computing. Many students, faculty, and practitioners in the industry highly recommend the courses. Bruegge et al. [Bru15] described their experience in supporting hundreds of students working with industrial clients. The authors concluded that integrating software engineering courses with real-world projects can improve students' technical and non-technical skills and prepare them for challenges they will face in their future careers.

---

<sup>1</sup><https://www.edx.org/bio/armando-fox>

## **1.4 Problems**

### **1.4.1 How to Get and Incorporate OSS Projects into Software-Engineering Courses?**

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 have larger course projects, focusing more on object-oriented design, and design patterns compared with CS1 or CS2 courses. One approach is to use existing OSS projects as the code base on which to build course projects.

Instructors who attempt to incorporate OSS projects into courses have to deal with three problems: (1) how to get ideas for OSS-based course projects with a reasonable amount of work, (2) how to provide consistent support during the semester, and (3) how to review students' deliverables. OSS core team members are best qualified to design course projects, offer technical support and review students' work. For instance, Ellis et al. [Ell07] 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.

### **1.4.2 How to Improve Student Contributions to OSS Projects?**

Unlike CS1/CS2 students who start small projects from scratch (normal course projects), students working on OSS-based course projects have to make contributions to an existing code base. Therefore, they have to consider how to make new code compatible with the current design. However, it is not easy for students to become familiar with their part of the project in such a short time period. Based on our research, we identify 13 mistakes that frequently occur in student contributions. Some mistakes occur not only in OSS-based course projects but also in normal course projects, such as commenting problems, bad naming, and hardcoding. Other mistakes are specific to OSS-based course projects, such as new code not following the existing design. Consequently, we need some ways to help students quickly understand the code base and make better contributions to OSS projects.



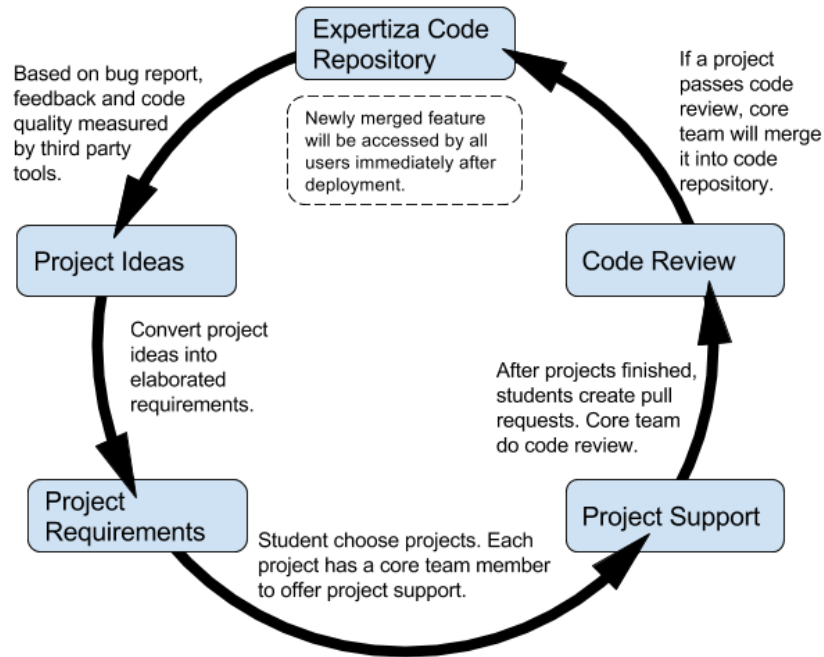


Figure 1.1: Expertiza development cycle

## 1.5 Expertiza

We maintain and run a student-authored open-source software application named Expertiza [Geh07]. It is an online peer-assessment tool initially funded by NSF. 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 [Son15b], peer-assessment rubric improvement [Son15a], calibrated peer assessment [Son16], and collusion detection in peer assessment [Son17]. The current Expertiza application was conceived in 2007. Since then, Expertiza has become the main source of course projects in our masters-level Object-Oriented Design and Development (OODD) course, CSC/ECE 517. The Ruby on Rails code base is available on GitHub.<sup>2</sup>

Figure 1.1 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 21 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

<sup>2</sup><https://github.com/expertiza/expertiza>

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, several widely-used tools will be triggered automatically to run all test cases and check the code quality according to Ruby Style Guide.<sup>3</sup> 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.

Over the years, more than 370 students have contributed code as deliverables through GitHub pull requests and helped Expertiza undergo several major updates. Below are two exemplars of student contributions that have extensively improved the performance and user interface (UI) of Expertiza.

Figure 1.2a shows the previous UI of the Expertiza content management page. We used Rails's built-in feature Embedded RuBy (ERB), to build the page. There is a search panel on the page, which allows instructors to filter and sort different kinds of content, such as courses, assignments, and questionnaires (rubrics and surveys). One big performance impact in the previous design is that each time an instructor executed a search query, the client had to fetch all data from the server again instead of requesting only the missing data. The page was redesigned with ReactJS by a team of students in 2014. The new UI in Figure 1.2b is much faster. Currently, if an instructor tries to filter or sort on different kinds of content, no more requests are sent to the server. All filtering and sorting happens at the client end. Students used flat UI elements to make this page more streamlined and efficient and make it responsive to changes in browser size across different devices.

Figure 1.3a illustrates the previous review and score report. At first glance, instructors can only figure out how many reviewers have reviewed this team's deliverables and the corresponding timestamps. Instructors have to click "show review" link to toggle the details of each review. This design is inefficient and instructors cannot compare different reviews directly. One team of students redesigned this report in 2015 by using a two-dimensional table shown in Figure 1.3b. The table shows the scores given for each question ("Criterion") in different rows and the scores given by each reviewer in different columns. What's more, instructors can hover over each cell to see the comments given by a particular reviewer. They can also compare scores and comments given by all reviewers on a question by clicking on each cell. Compared with the previous UI, the new one combines more information into a similar space on the page. This team of students also optimized the back-end data processing logic to reduce the page loading time.

---

<sup>3</sup><https://github.com/bbatsov/ruby-style-guide/blob/master/README.md>

Expertiza

Reusable learning objects through peer review

Home

Manage...

Survey Deployments

Assignments

Course Evaluation

Profile

Contact Us

Home

User: user6

Logout

Manage content

Search:

in

Questionnaire

Search

Reset

Filter:

Assignment by course name

Filter

Sort by:

date created

descending

Sort

Details		Actions
+ ↗	Questionnaires	
+ ↗	Courses	
- ↗	Assignments	
	<div>New problems B</div> <div>Directory: madeup/b</div> <div>Creation Date: 2014-03-05 14:21:59 UTC</div> <div>Updated Date: 2014-10-18 01:11:58 UTC</div>	

(a) Previous UI

expertiza

Home

Manage...

Survey Deployments

Assignments

Course Evaluation

Profile

Contact Us

User: Instructor 6

Impersonate...

Revert

Logout

Manage content

Courses

Assignments

Questionnaires

Search...

☐

Include others' items

New public course

New private course

Name↑↓	Institution↑↓	Creation Date↑↓	Updated Date↑↓	Actions
My Courses				
Object-Oriented Design and Development, Fall 2018	-	Sep 1, 2018 - 11:05 PM	Sep 1, 2018 - 11:05 PM	
Object-Oriented Design and Development, Spring 2018	-	Jan 19, 2018 - 11:25 PM	Jan 19, 2018 - 11:25 PM	
Architecture of Parallel Computers, Fall 2017	-	Aug 29, 2017 - 11:49 PM	Sep 23, 2017 - 2:47 PM	
Object-Oriented Design and Development, Fall 2017	-	Aug 29, 2017 - 11:49 PM	Sep 20, 2017 - 11:28 PM	
Object-Oriented Design and Development, Spring 2017	-	Jan 13, 2017 - 11:03 PM	Jan 13, 2017 - 11:03 PM	

(b) Current UI

Figure 1.2: Screenshots of content management pages

## Review

<b>Review 1</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 1	<b>Last Reviewed:</b> Wednesday November 09 2016, 05:49PM
<b>Review 2</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 2	<b>Last Reviewed:</b> Monday November 07 2016, 09:10AM
<b>Review 3</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 3	<b>Last Reviewed:</b> Wednesday November 09 2016, 01:24PM
<b>Review 4</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 4	<b>Last Reviewed:</b> Friday November 04 2016, 02:05PM
<b>Review 5</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 5	<b>Last Reviewed:</b> Tuesday November 08 2016, 12:59AM
<b>Review 6</b> <a href="#">show review</a> <a href="#">Give feedback</a> for Review 6	<b>Last Reviewed:</b> Wednesday November 09 2016, 04:09PM

(a) Previous UI

Review	<a href="#">Toggle Question List</a>	Color Legend		Interaction Legend			
Criterion		Review 1	Review 2	Review 3	Review 4	Review 5	Review 6
1	✓	0	0	0	0	0	0
2	⑤		3	3	3	0	3
3	⑤	5	5	5	5	0	2
4	⑤	3	5	4	5	0	2
5	⑤	3	5	0	5	0	2
6	⑤	5	4	3	5	3	5
7	⑤	3	1	0	4	0	1
8	⑤	4	5	4	5	0	5
9	🔍						

(b) Current UI. The first row lists reviewer names. The student view uses “Review X” in place of reviewer names. The first column displays the number of questions (“Criteria”) with their corresponding types: (1) a check mark represents a checkbox question; (2) a number in a circle represents a Likert scale question (here represents a 1–5 Likert scale question); (3) a pencil mark represents a paragraph (free text) question. The background colors of cells are scaled from poor to excellent in the following order: red, orange, yellow, light green, and dark green. And gray indicates that the reviewer did not give a score for a particular question. A score in an underlined bold font means that there are corresponding comments on this question.

Figure 1.3: Screenshots of review and score reports

## 1.6 Course Settings

The OODD course has been opened in North Carolina State University for more than 25 years. Students come from different countries, but with a predominance from India, usually more than 70%. The other prominent countries of origin are China and the United States. 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, most of them Expertiza-based projects (other projects have been based on Mozilla, Sahana, Apache, and OpenMRS, among others). 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 100–200 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 100–200 lines of testing code.

All course projects are done in teams. The desired team size for the initial course project is 2–3, and for the final course project is 3–4. To give each student a diversity of development 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 [Akb18]. Students are asked to choose which project topics they are interested in, and are allowed (but not required) to form teams before indicating their preference. They register their preferences in Expertiza, and the algorithm matches students and teams with project topics. Students are allowed to switch teams and project topics if they don't like the team or topic they have been assigned.

After students finish course projects, we review their deliverables. If they have done a good job, we merge their contributions into the code repository. If the entire project is not acceptable, we may also partially merge their contributions, that is, we merge particular parts of students' code directly into the code repository, and refactor or remove the remaining part. In the worst case, we reject student contributions. We usually merge projects whose score is  $\geq 93/100$ . In the other two situations, the grade will normally be lower than 93.

## 1.7 Overview of the Approach

In this dissertation, we address the problem of *helping students make better contributions to open-source software projects*. The following is the structure of this dissertation.

Chapter 2 shares our long-term experience on how to maintain Expertiza with limited resources. We discuss what challenges instructors would meet when they attempt to maintain and run a software application implemented by students and what approaches they could adopt to overcome these issues.

Chapter 3 shares our experience in supporting nearly 1000 students on 461 OSS-based course projects from 2012 to 2018. We manually checked 313 course projects from the 2012 fall semester to the 2017 spring semester, and we summarize 13 mistakes that frequently occur in student contributions. Based on the criteria for each common mistake, we further tallied common mistakes in 148 course projects from the 2017 fall semester to the 2018 fall semester. We then give four suggestions to help students reduce the frequency of common mistakes and improve the quality of their OSS pull requests.

Chapter 4 presents our effort to conduct a quasi-experimental controlled study to test the efficiency of the test-driven approach on OSS-based course projects. Thirty-five masters students participated in our study. Students worked on course projects in teams, half of which were assigned to the TDD group (using the test-driven approach), and the rest of which were assigned to the non-TDD group (using the traditional test-last approach). We found that the test-driven approach can help students in the TDD group write more tests and achieve significantly higher statement coverage than students in the non-TDD group.

Chapter 5 discusses how we bypass the limitations of existing third-party tools by implementing three Internet bots, namely the Danger Bot, the Travis CI Bot, and the Code Climate Bot. One-hundred one masters students participated in our study. We found that bots can provide more timely feedback than teaching staff, and significantly reduce system-specific guideline violations and code smells in student contributions.

Chapter 6 concludes this report with a summary of the work and directions for future research.

## CHAPTER

# 2

# LONG-TERM MAINTENANCE OF STUDENT-AUTHORED OPEN-SOURCE SOFTWARE

## 2.1 Introduction

In this chapter, we share our long-term experience on how to maintain a student-authored open-source software with limited resources. During these years, we have maintained a user base for Expertiza. 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.

## 2.2 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

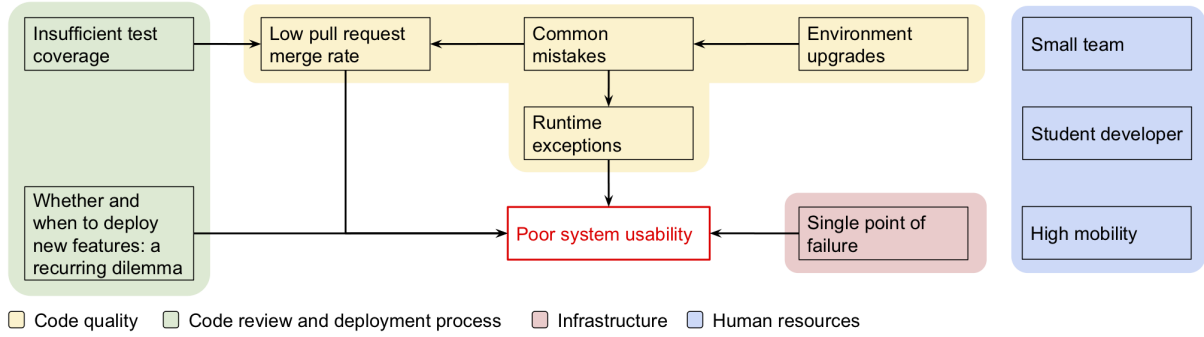


Figure 2.1: Relationship among different perspectives

Table 2.1: Challenges and approaches in each perspective

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

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 2.1 shows the relationship between 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 2.1 summarizes challenges and corresponding approaches in each perspective. We discuss them in detail below.



### 2.2.1 Code Quality

The first challenge instructors face is how to maintain a high-quality code base. If the software 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.<sup>1</sup> 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 [Hu18]. 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 research (shown in Chapter 3), we manually checked 313 course projects from 2012 to 2017 and summarized 13 **common mistakes** that frequently occur in students' deliverables [Hu18]. The proportion of the five most frequent mistakes—which include commenting, shallow or missing tests, bad naming, duplicated code, and failing functionality—exceeds 50% in most semesters. 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 **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,<sup>2</sup> which is free

---

<sup>1</sup><https://codeclimate.com/blog/abandoned-pull-requests/>

<sup>2</sup><https://codeclimate.com/github/expertiza/expertiza>

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 (shown in Chapter 5). Our pilot study [HG19] 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<sup>3</sup> 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.

### 2.2.2 Code Review and Deployment Process

Our typical code review and deployment pipeline consists of six steps as shown in Figure 2.2: (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,<sup>4</sup> 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.

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 [Hu18], which weakens their fault-finding capability. Before

---

<sup>3</sup><https://airbrake.io/>

<sup>4</sup>We use Travis CI as a continuous integration service. See <https://travis-ci.org/expertiza/expertiza>.

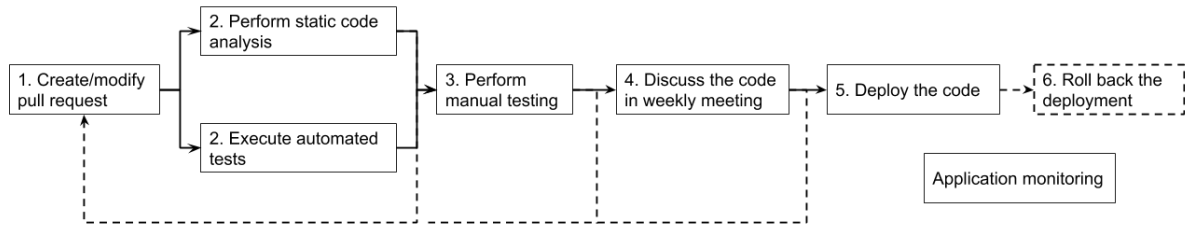


Figure 2.2: Code review and deployment pipeline

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<sup>5</sup> 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<sup>6</sup>, 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

<sup>5</sup><https://coveralls.io/github/expertiza/expertiza?branch=master>

<sup>6</sup><https://www.ansible.com/>

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<sup>7</sup> 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 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.<sup>8</sup> If developers encounter some runtime errors in the green environment, they can easily switch back to the blue environment.

### 2.2.3 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.

### 2.2.4 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

---

<sup>7</sup><https://capistranorb.com/>

<sup>8</sup><https://martinfowler.com/bliki/BlueGreenDeployment.html>

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.

## **2.3 Conclusion**

We have shared our long-term 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.

## CHAPTER

# 3

# COMMON MISTAKES IN STUDENT CONTRIBUTIONS TO OPEN-SOURCE SOFTWARE PROJECTS

## 3.1 Introduction

In this chapter, we discuss our experience 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. Based on the criteria for each common mistake, we further tallied common mistakes in 148 course projects from the 2017 fall semester to the 2018 fall semester. We detected that the relative occurrence of the five most frequent mistakes, including commenting, shallow or missing tests, bad naming, duplicated code, and failing functionality, has remained stable across last 12 semesters. These account for more than 50% of mistakes each semester. We also found that rejected projects had significantly more problems with missing pull requests/not forked and failing functionality than partially merged projects. Moreover, in another aspect, final projects tended to have significantly more of almost all common mistakes than initial projects. Then we give four suggestions to help students to reduce common mistakes and achieve OSS pull requests with higher quality.

## 3.2 Related Work

Many papers talk about incorporating OSS projects into computer science education. Bishop et al. [Bis16] 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 [Geh07; RK06; Ell12; McC12; Liu05].

Some studies discuss the benefits of integrating OSS projects into software-engineering courses. Ellis et al. [Ell12] 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 & Kazemian [RK06] 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. [Pin17] 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 [Tot06] postulated that the first challenge of using OSS projects in software-engineering courses is to identify and select appropriate OSS projects. Smith et al. [Smi14] 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 [Geh11] 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. [Ell08] 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 student contributions to OSS projects. In their literature review paper, Nascimento et al. [Nas15] 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 student contributions to OSS projects. Buchta et al. [Buc06] introduced a detailed grading rubric involved in implementation of functionality, interaction with the project manager, the format of the report, and so on. Liu [Liu05] 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 studies talking about common mistakes that frequently occur in student contributions to OSS projects. In this chapter, we summarize 13 common mistakes organized into six categories, which could become a checklist to help evaluate students' code.

### 3.3 Expertiza and Other OSS Projects

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 [Ell07], 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 [Bal02]. The core team of Expertiza has three members right now, which is composed of one professor, one Ph.D. student, and one undergraduate student. 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. Student 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.

### 3.4 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 3.1). 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. We (I and my colleague Yang Song) performed *open coding*<sup>1</sup>—we went through the project grades and feedback, GitHub pull request code, and comments on 221 rejected

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Open\\_coding](https://en.wikipedia.org/wiki/Open_coding)



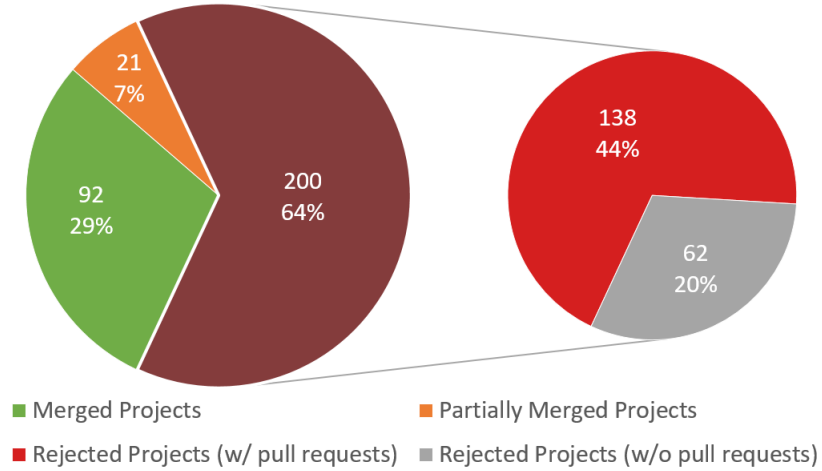


Figure 3.1: Merge status of OSS-based course projects from the 2012 fall semester to the 2017 spring semester

or partially merged projects (dark red and orange),<sup>2</sup> created a new label whenever we detected a new common mistake, and tagged projects using existing labels. During the coding process, we implemented the technique of *negotiated agreement* to code reliably [Cam13].

We identified a total of 13 common mistakes. 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. [Gou15], 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.

### 3.4.1 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

<sup>2</sup>We assumed that there were no common mistakes in merged projects.

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. **Commenting** 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.

### 3.4.2 Common Coding Mistakes

An important rule of software design is the DRY principle: “Don't repeat yourself” [HT00]. If one feature or one piece of code is presented multiple times in one system, anyone who refactors 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 re-implement functionality without bothering to do a thorough search to see if it 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 3.2 (once in the controller and twice in the models), which caused the topic re-assignment functionality to be difficult to maintain.

**Long methods** 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

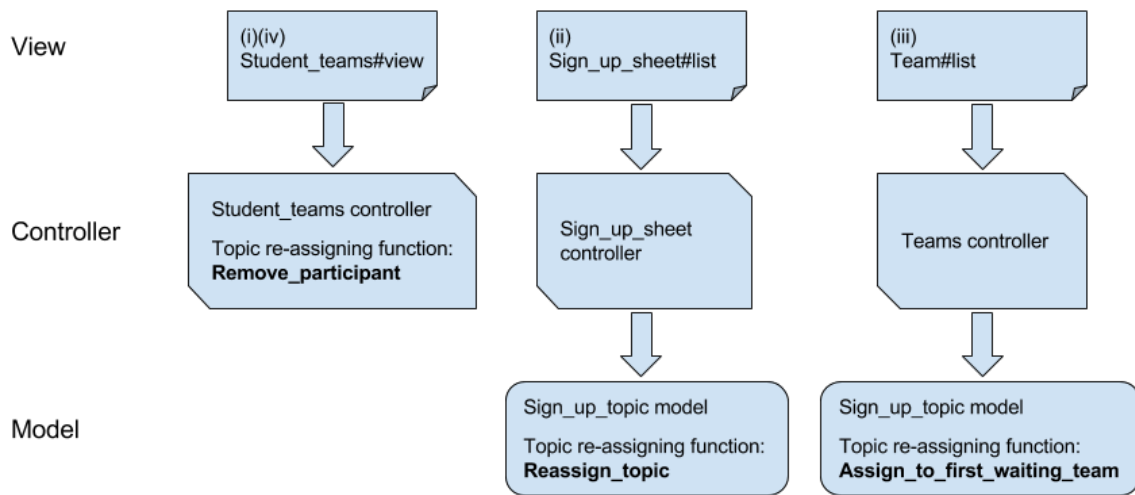


Figure 3.2: Functionality for reassigning topics in Expertiza. The left flow chart shows the functionality for scenario (i) and (iv). The middle chart displays the functionality for scenario (ii). The right one presents the functionality for scenario (iii).

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**,<sup>3</sup> you should think of polymorphism.

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, failing to implement some requirements of the project, and then bypassing them during the demo.

### 3.4.3 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.

<sup>3</sup><https://sourcemaking.com/refactoring/smells/switch-statements>

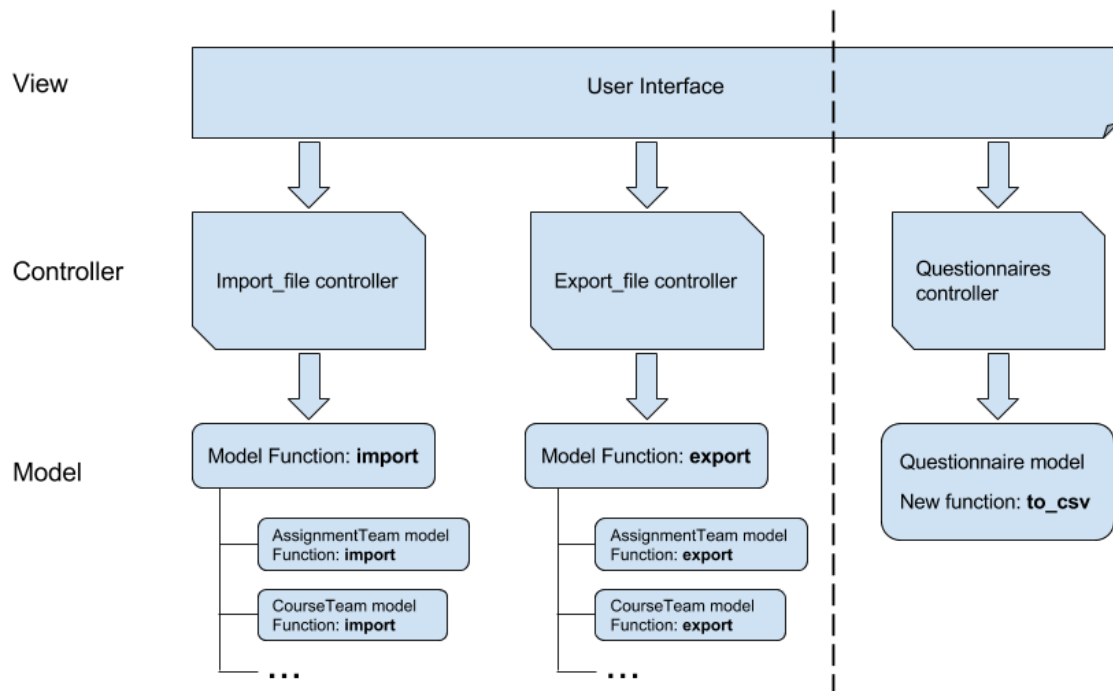


Figure 3.3: Functionality for importing/exporting questionnaires in Expertiza

```

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

```

Figure 3.4: An example of shallow test

One good example is the OSS-based course project to fix the functionality of importing and exporting questionnaires. Figure 3.3 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 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.

### 3.4.4 Common Testing Mistakes

Pull requests with tests have a higher chance to be merged [Tsa14]. 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**. 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 3.4 shows one RSpec<sup>4</sup> 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.

### 3.4.5 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

---

<sup>4</sup><http://rspec.info/>

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 their fork with 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.

Back in 2012, we found that many students did **not create pull requests**. This made it difficult for core team members to review the students' work. A more serious situation is when students had **not even forked** the Expertiza repository. Instead, they 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 historical information and the newly-created repository cannot be associated with the Expertiza repository, 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.

#### 3.4.6 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 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.

### 3.5 Analysis of Common Mistakes

Table 3.1 summarizes the criteria for each common mistake. Based on those, we tallied common mistakes in course projects from recent three semesters, namely the 2017 fall semester, the 2018 spring semester, and the 2018 fall semester. There were a total of 461 OSS-based course projects related to Expertiza from the 2012 fall semester to the 2018 fall semester: 126 projects (27%) were merged into the Expertiza code base; 41 projects were partially merged; 294 projects were rejected, including 63 projects that did not submit pull requests.

We then analyze these common mistakes in different dimensions. First, we present an overview of common mistakes by identifying the occurrence of each common mistake in course projects. Then we conduct statistical analyses on each common mistake in rejected vs. partially merged projects, as well as in initial vs. final projects to figure out whether common mistakes are more

Table 3.1: Criteria for common mistakes

Common mistakes	Criteria for identifying common mistakes
Commenting	<ul style="list-style-type: none"> <li>- Missing comments for newly-added/modified methods.</li> <li>- Writing comments that repeat the method names or variable names.</li> <li>- Writing inline comments for each line of code.</li> <li>- Leaving commented debug code in pull requests.</li> <li>- Commenting out code that needs to be deleted.</li> </ul>
Shallow or missing tests	<ul style="list-style-type: none"> <li>- Not writing tests for the newly-added/modified code.</li> <li>- Including skipped/pending/focused test cases.</li> <li>- Including greater than or equal to five wildcard argument matchers (e.g., <code>anything</code>, <code>any_args</code>).</li> <li>- Not writing/commenting out expectations for the tests.</li> <li>- Using test expectations that lack matchers, such as comparisons (e.g., <code>equal(expected_value)</code>), the status change of objects (e.g., <code>change(object, :value).by(delta)</code>), error handlings (e.g., <code>raise_error("message")</code>).</li> <li>- Using test expectations that only focus on the return value not being <code>nil</code>, <code>empty</code>, or not equal to 0 without testing the <i>real</i> value.</li> <li>- In feature tests, using expectations that only focus on words appearance on the view (e.g., <code>expect(page).to have_content(word)</code>), and without other evidence, such as the new creation of the object, new record in database.</li> </ul>
Bad naming	<ul style="list-style-type: none"> <li>- Using meaningless method/variable names, such as <math>a</math>, <math>v_1</math>.</li> <li>- Using method/variable names that not follows the naming convention of the programming language (e.g., for Ruby, use <i>snake_case</i> for symbol, variable and method names; use <i>CamelCase</i> for class and module names.).</li> </ul>
Duplicated code	<ul style="list-style-type: none"> <li>- Newly-added/modified code not following the DRY principle.</li> <li>- Implementing methods which duplicate the existing functionality.</li> </ul>
Failing functionality	<ul style="list-style-type: none"> <li>- Refactoring or new features that do not pass automated/manual tests.</li> </ul>
Messy pull requests	<ul style="list-style-type: none"> <li>- Committing many debugging code.</li> <li>- Committing many unnecessary configuration file (e.g., database schema file) changes.</li> <li>- Committing many white space or indentation changes.</li> <li>- Committing files that should be ignored (placed in <code>.gitignore</code> file).</li> <li>- Committing many third-party library files.</li> <li>- Committing modifications from other projects.</li> </ul>
Hardcoding	<ul style="list-style-type: none"> <li>- Inserting data directly into the source code instead of obtaining the data from external sources or creating it at run time.<sup>a</sup></li> </ul>
Not following the existing design	<ul style="list-style-type: none"> <li>- The submission not following the existing design of the system.</li> </ul>
Missing pull request/ not forked	<ul style="list-style-type: none"> <li>- Not creating a pull request.</li> <li>- Downloading the code, creating a new repository and uploading the modified code, instead of forking the repository.</li> </ul>
Long methods	<ul style="list-style-type: none"> <li>- A method that exceeds 60 LoC.</li> </ul>
Bad specification	<ul style="list-style-type: none"> <li>- Inaccurate project requirements.</li> <li>- The submission that deviates far from requirement due to the insufficient communication between teaching staff and students.</li> </ul>
Switch statements	<ul style="list-style-type: none"> <li>- Using big <code>switch</code> or <code>if</code> statements instead of adopting polymorphism or other mechanisms.</li> </ul>
Limited modifications	<ul style="list-style-type: none"> <li>- The submission that changes less than 50 LoC.</li> </ul>

<sup>a</sup> [https://en.wikipedia.org/wiki/Hard\\_coding](https://en.wikipedia.org/wiki/Hard_coding)

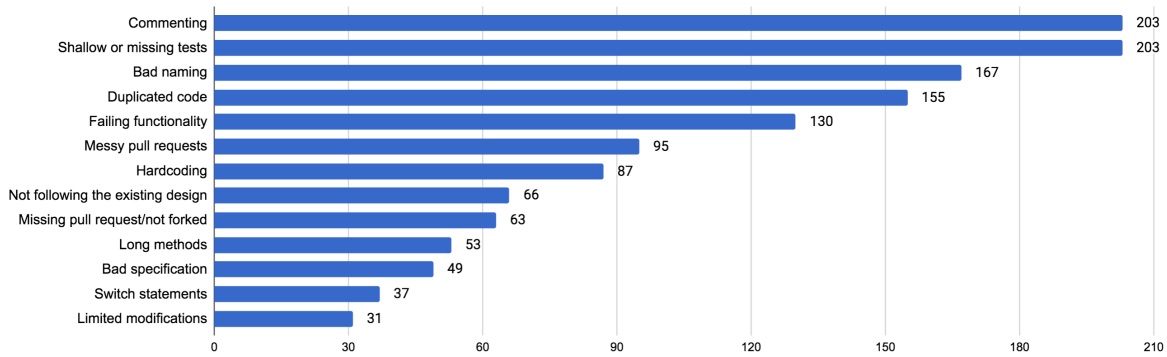


Figure 3.5: Number of course projects that has each common mistake

frequent in rejected vs. partially merged projects, or in initial vs. final projects. Instead of counting the frequency, of each common mistake in course projects, we merely tally the existence or absence of each mistake. This is because it is essentially impossible to determine how many times, e.g., the existing design is not followed.

### 3.5.1 Overview of Common Mistakes

We tallied the occurrence of each common mistake in 335 rejected or partially merged projects shown in Figure 3.5. Among all rejected or partially merged projects, more than 60% have commenting and testing problems, and almost half of them have naming problems. Other common mistakes like duplicated code and failing functionality also occurred with a high frequency. We also used the treemap method (shown in Figure 3.6) to visualize the number of course projects for each common mistake per semester. Each color represents one semester. Each rectangle shows the name of a common mistake and the number of projects with this common mistake. Note that the area of rectangles is proportional to the *number* of mistakes. Table 3.2 shows the number of students enrolled in our course per semester. The number of students (and projects) is greater in fall semesters than spring semesters. As popularity grows, a larger number of common mistakes occurs.

Figure 3.7 shows the frequency of occurrence of each common mistake in each semester. We identified the five most common mistakes from Figure 3.5 and aggregated other common mistakes into the “Other” category. The actual number of course projects that have each mistake is displayed in the middle of each rectangle. According to Figure 3.7, there is a big difference between the 2012 fall semester and other semesters, considering that back in 2012, many projects did not create pull requests. Since 2013, the proportion of the five most common mistakes has remained stable across most semesters. Together they represent more than half of all common mistakes in each semester.



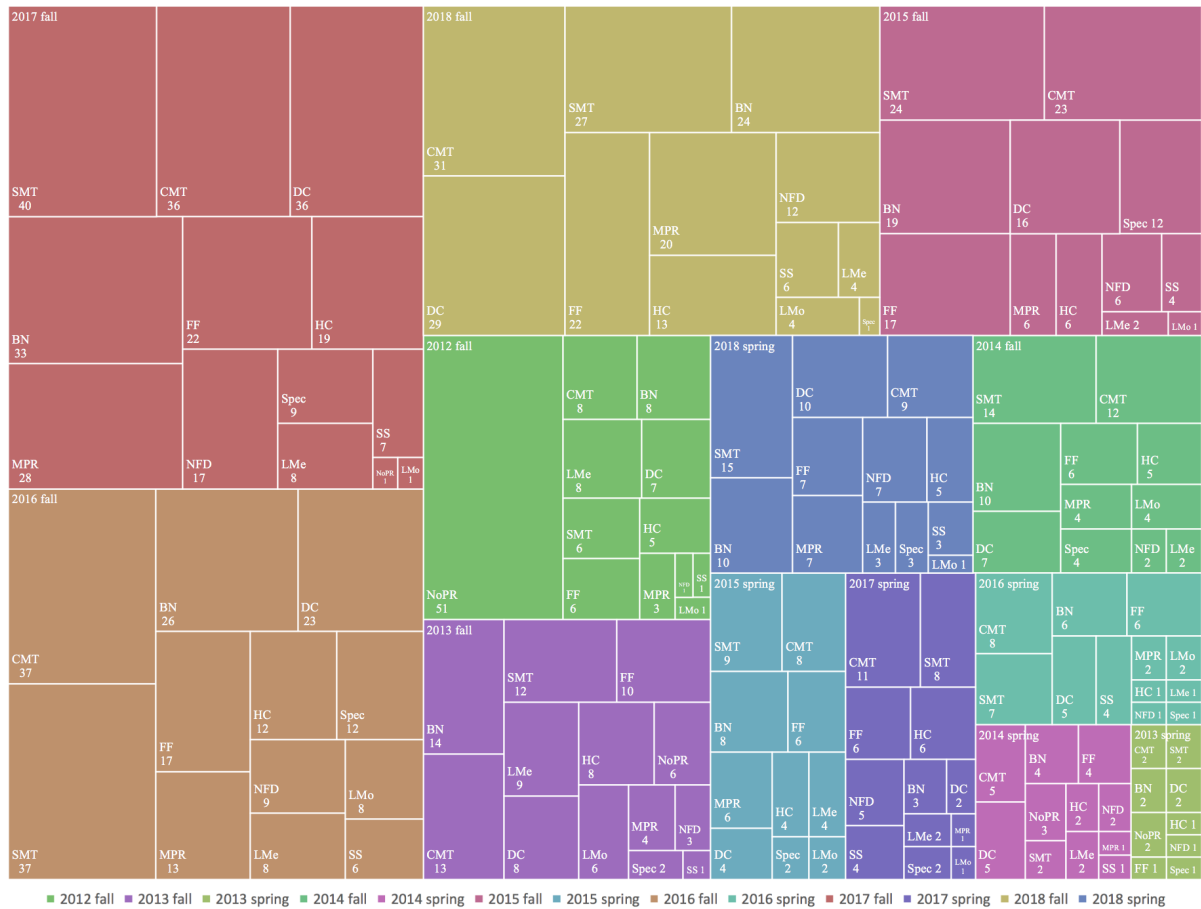


Figure 3.6: Number of course projects that have each common mistake, by semester. Due to space limitations, we abbreviate each common mistake: CMT (commenting), SMT (shallow or missing tests), BN (bad naming), DC (duplicated code), FF (failing functionality), MPR (messy pull request), HC (hardcoding), NFD (not following the existing design), NoPR (missing pull request/not forked), LMe (long methods), Spec (bad specification), SS (switch statements), and LMo (limited modifications).

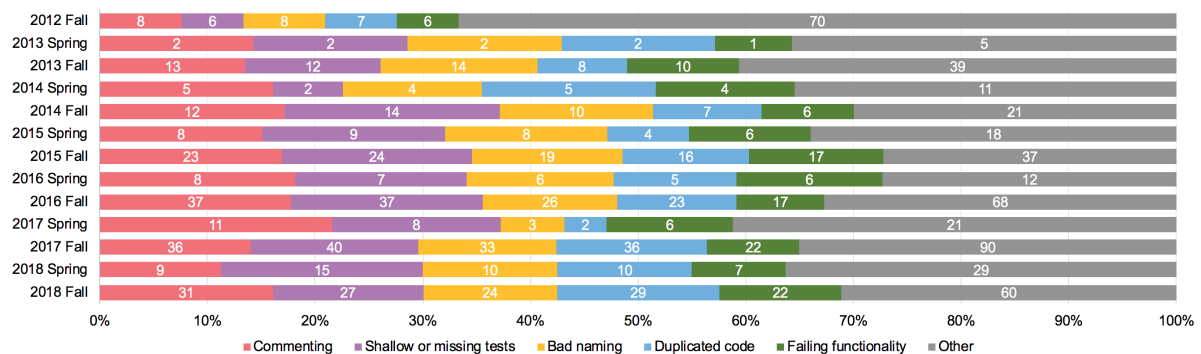


Figure 3.7: Frequency of occurrence of most-common mistakes, by semester

Table 3.2: Number of students enrolled in each spring (S) and fall (F) semester

Semester	12F	13S	13F	14S	14F	15S	15F	16S	16F	17S	17F	18S	18F
# of students	138	18	71	19	78	36	95	53	132	62	120	49	106

### 3.5.2 Common Mistakes in Rejected vs. Partially Merged Projects

We first applied a one-sided Wilcoxon rank-sum test to check whether the occurrence of all common mistakes in 41 partially merged projects was significantly less than in the 294 rejected projects ( $p = 0.027 < 0.05$ ). We then conducted Chi-square tests on rejected and partially merged projects to determine whether these two sets of projects differed on each common mistake. Table 3.3 displays contingency tables for Chi-square tests. Results indicated that there were significant differences in failing functionality and missing pull request/not forked issues for rejected and partially merged projects.

We further studied these two common mistakes. About 41% of rejected projects and 22% of partially merged projects have a failing functionality problem. Moreover, more than 21% of rejected projects but no partially merged projects had missing pull request/not forked issues. Because of the Chi-square results, we concluded that rejected projects tended to have more failing functionality and missing pull request/not forked problems than partially merged projects. We thought that the occurrence of these two common mistakes could help us decide whether to reject a particular project and speed up our grading process: when grading a project, we could first check whether they exhibit missing pull request/not forked or failing functionality issues in student contributions; then we could decide whether to continue to look at other common mistakes.

### 3.5.3 Common Mistakes in Initial vs. Final Projects

We are also interested in whether common mistakes occurred more in initial or final projects. Similar to before, we performed a one-sided Wilcoxon rank-sum test. The result ( $p = 7.2e^{-8} < 0.05$ ) revealed that the occurrence of common mistakes in final projects was significantly more than initial projects. We also conducted Chi-square tests to examine the differences between initial and final projects for each common mistake. Table 3.4 displays contingency tables for Chi-square tests. Results indicated that final projects tended to have significantly more of almost all common mistakes, except failing functionality, missing pull request/not forked, and limited modifications. Conversely, initial projects had significantly more problems of limited modifications. This may be because initial projects often focus on refactoring the existing code base, while final projects frequently involve designing and implementing new features and changing more LoC. The complexity of final projects may make students think a lot about the functionality and ignore the design, code style, testing, and

Table 3.3: Chi-square tests for common mistakes in rejected vs. partially merged projects

Common mistakes	Has this mistake?	# of rejected projects	# of partially merged projects	P value of Chi-square test
Commenting	Yes	176	27	0.57
	No	118	14	
Shallow or missing tests	Yes	179	24	0.91
	No	115	17	
Bad naming	Yes	147	20	0.98
	No	147	21	
Duplicated code	Yes	133	21	0.58
	No	161	20	
Failing functionality	Yes	120	9	<b>0.031*</b>
	No	174	32	
Messy pull requests	Yes	83	12	0.96
	No	211	29	
Hardcoding	Yes	82	5	0.050
	No	212	36	
Not following the existing design	Yes	58	8	0.86
	No	236	33	
Missing pull request/ not forked	Yes	63	0	<b>0.0021*</b>
	No	231	41	
Long methods	Yes	50	3	0.17
	No	244	38	
Bad specification	Yes	43	4	0.55
	No	251	37	
Switch statements	Yes	35	2	0.28
	No	259	39	
Limited modifications	Yes	26	5	0.68
	No	268	36	

\* The significance level is 5%.

communication with OSS core team members.

Table 3.4: Chi-square tests for common mistakes in initial vs. final projects

Common mistakes	Has this mistake?	# of initial projects	# of final projects	<i>P</i> value of Chi-square test
Commenting	Yes	106	97	<b>0.012*</b>
	No	88	44	
Shallow or missing tests	Yes	108	95	<b>0.040*</b>
	No	86	46	
Bad naming	Yes	85	82	<b>0.013*</b>
	No	109	59	
Duplicated code	Yes	77	77	<b>0.0095*</b>
	No	117	64	
Failing functionality	Yes	67	62	0.10
	No	127	79	
Messy pull requests	Yes	46	49	<b>0.037*</b>
	No	148	92	
Hardcoding	Yes	33	54	<b>2.0e<sup>-5</sup>*</b>
	No	161	87	
Not following the existing design	Yes	26	40	<b>0.0011*</b>
	No	168	101	
Missing pull request/ not forked	Yes	43	20	0.088
	No	151	121	
Long methods	Yes	21	32	<b>0.0053*</b>
	No	173	109	
Bad specification	Yes	18	29	<b>0.0055*</b>
	No	176	112	
Switch statements	Yes	15	22	<b>0.036*</b>
	No	179	119	
Limited modifications	Yes	24	7	<b>0.034*</b>
	No	170	134	

\* The significance level is 5%.

### 3.6 Quality Control Suggestions for OSS Pull Requests

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

### E1713. Refactor `penalty_helper.rb` and `late_policies_controller.rb`

**Contact:** Ed Gehringer ([efg@ncsu.edu](mailto:efg@ncsu.edu))

**What they do:** These classes implement late penalties in Expertiza. If an action (submission, review, etc.) is performed after the deadline, Expertiza can be set up to deduct points automatically. Read about this functionality [here](#). You can also read about the penalty wizard and associated tests [here](#).

**What's wrong with it:** The biggest problem with these files is that the methods are too long; they do multiple things. They should be broken up into simpler methods.

**Files involved:** [penalty\\_helper.rb](#), [late\\_policies\\_controller.rb](#)

**What needs to be done:**

- Break up methods into no more than about 25 lines each. At the beginning of each method, write a method comment saying what the method does.
- Write functional tests for the methods you create. You may be able to find and modify the previous functional tests, which were included in the [Github repository](#).
- In many places, there is no space before “(if ... )”; insert a space. In programming, as well as English prose, an open parenthesis should *always* be preceded by a space unless it begins a parameter list. And a close parenthesis should always be followed by a space, unless it is immediately followed by punctuation. But programmers don't violate that rule so often.

Figure 3.8: An example of well-designed project tasks

#### 3.6.1 Well-designed Project Tasks

Figure 3.8 is an example of well-designed project tasks. Based on our experience, a good project requirement statement should consist of four parts: (1) Background information (“What they do” in Figure 3.8), for instance, why do we need this new feature, what is the problem with the current design? etc. (2) Overall requirements (“What's wrong with it”), explicitly telling students at a high level what they need to achieve. (3) A list of the main files involved in this project (“Files involved”), to show students where to start the project. (4) Detailed requirements (“What needs to be done”), which will directly guide students through the project requirements.

#### 3.6.2 Better Communication between Core Team Members and Students

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.

### 3.6.3 Test-Driven Development

Test-driven development is one way to develop software by turning requirements into specific test cases.<sup>5</sup> Students are asked to convert each task into failed test cases, make failed test cases 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.

### 3.6.4 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, we can adopt third-party tools, such as Travis CI, Code Climate, Coveralls, Internet bots, to automatically check the quality of student contributions and provide instant feedback.

## 3.7 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, well-designed project tasks, and review student 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 implicitly demonstrate 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

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development)

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 follow the existing design.

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 problems. Others like “not following the existing design” may happen more frequently in OSS-based course projects. In this chapter, 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.

## **3.8 Threats to Validity**

### **3.8.1 Internal Validity**

We have introduced several mechanisms in recent semesters, such as more attention to testing and coding style and weekly meetings between mentors and student teams, that may improve student performance on course projects and reduce the number of common mistakes.

### **3.8.2 External Validity**

Our 13 common mistakes are based on analysis of Expertiza pull requests. These common mistakes may not be generalizable to other projects. However, we have analyzed more than 450 pull requests from 2012 to 2018, which might mitigate this threat.

### **3.8.3 Construct Validity**

We assumed that there were no common mistakes in merged projects. Hence, we did not check them. It is possible that there still exist some (non-fatal) common mistakes in these projects. Also, we utilized the technique of negotiated agreement when coding common mistakes. Therefore, we were not able to calculate inter-rater reliability. This may be a threat to construct validity.

## **3.9 Conclusions**

In this chapter, 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 to accept or reject a particular pull request. We found that the relative occurrence of the five most frequent mistakes has remained stable across several semesters. Final projects tended to have significantly more of almost all common mistakes than initial projects, but this is because final projects make more substantial code modifications than initial projects.

Furthermore, we give four 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.



## CHAPTER

# 4

# A TEST-DRIVEN APPROACH TO IMPROVING STUDENT CONTRIBUTIONS TO OPEN-SOURCE SOFTWARE PROJECTS

## 4.1 Introduction

The history of test-driven development (TDD) can be traced back to the 1950s. D.D. McCracken introduced the concept of TDD in his 1957 book *Digital Computer Programming* [McC57]. But TDD was not widely utilized until Kent Beck “rediscovered” the technique in 2003. In his book *Test-Driven Development*, Beck introduced the TDD mantra—“red/green/refactor” [Bec03]. Since then, the test-driven approach has matured to a formal iterative process where developers (1) write failing tests for the code they are about to develop, (2) implement functionality to make the tests pass, and (3) refactor out any duplicated code. In another scenario, if initial tests pass, a developer can start refactoring existing code directly and keep the tests passing during development. The major benefits of this process are that it gives developers confidence in the soundness of their work, and it eases the burden of refactoring by quickly verifying that functionality has not changed.

Given the potential benefits of TDD, educators introduced it into many computer-science courses to help students write high-quality code with fewer defects [Edw03b; SP06; JS08; Jon04; Erd05]. Several researchers conducted structured experiments and evaluated the effectiveness of

the test-driven approach for entry-level computer science courses [PC11; Cau12b]. Some experimental results showed that the test-driven approach can help developers write high-quality code and improve design skills [GW04; Des09]. But other researchers reached opposite conclusions, namely, that the test-driven approach does not always produce high-quality code, especially when used by inexperienced developers [SA07; PC11]. There is evidence that, compared with traditional development methods, the test-driven approach is no more effective in eliminating defects [Will12]. However, previous experiments were based on throwaway projects—projects used one time and discarded once finished [Hu18].

This chapter reports on a quasi-experimental controlled study based on Expertiza. It has more than 30,000 lines of source code, which makes Expertiza-based course projects potentially more complex than throwaway projects and closer to real-world projects. In the course of the experiments, we had the students work on existing files with very low statement coverage or no tests at all, to ensure that students had ample opportunity to write tests first before refactoring the codebase. Then we implemented *test skeletons* to help students write tests. When student contributions are merged into the current codebase, they benefit all Expertiza users. Also, these projects can benefit students in their future careers by giving them experience with OSS projects and test-driven approach.

We found that students were able to comply with the test-driven approach for more than 20% of the duration of the project. There were no major differences between the TDD group and the non-TDD group in the quality of source-code modifications and newly-created tests. Yet the TDD group wrote more tests and achieved significantly higher statement coverage than the non-TDD group—12% more statement coverage, on average.

## **4.2 Related Work**

### **4.2.1 Introducing the Test-Driven Approach to Computer Science Education**

Previous researchers have studied the effectiveness of the test-driven approach in computer-science education. Many experimental results showed that programming submissions in entry-level courses have fewer defects when the test-driven approach is used. Edwards [Edw03a] is one of the pioneers of using the test-driven approach in education. Since 2003, his web-based automated testing tool called Web-CAT has helped undergraduates use the test-driven approach. The results indicated that code written by students using Web-CAT has fewer defects [Edw03b].

Since then, more researchers have explored the benefits of introducing test-driven approach in computer education. Most researchers took undergraduates as research subjects and conducted experiments in entry-level courses. Desai et al. [Des09] did two controlled experiments in CS1/CS2 courses. They found that although the workload may increase after implementing the test-driven approach, students were able to successfully complete the unit tests. Janzen & Saiedian [JS06] also

did TDD experiments with undergraduates in a software engineering course. Their results showed that the test-driven approach is an effective way to improve code quality and increase the confidence of developers.

Few researchers have studied the test-driven approach with more advanced students. Kollanus & Isomöttönen [KI08] conducted two experiments on TDD in a masters-level course and discussed their experience. The first assignment was to implement a simple HTTP server, and the second one was to write a small software component, which can be used for text processing. They summarized difficulties that students had with TDD and claimed that their assignments were relatively small and were inconsistent with the reality of software engineering work. Causevic et al. [Cau12b] did a pilot experiment for masters students in 2012. The object of their experiment was a bowling game score calculation problem, which was based on the *Bowling Game Kata*.<sup>1</sup> They found that test cases created by the test-driven approach and the traditional test-last approach have almost the same quality.

#### 4.2.2 Measurement of Compliance with the TDD Process

Many researchers have conducted experiments to test the effectiveness of the test-driven approach [JS06; Cau12a; XL09; GW03; Can06; Erd05; FT13]. They have analyzed source code, as well as survey and interview data. However, only a few papers mention mechanisms to measure subjects' compliance with the TDD process, which is crucial to the validity of controlled experiments.

Buffardi & Edwards [BE12] asked students to submit their work several times. Then they were able to review snapshots of the work and analyzed students' development process. However, these snapshots of work cannot indicate whether the tests were written before or after the corresponding source code. Moreover, they used self-reported adherence to TDD to verify their hypotheses, which may be unreliable. Müller & Höfer [MH07] developed a system that uses two Eclipse plugins called *user-action logger* and *JUnit-action logger*. The user-action logger captures focus changes among different parts of the Eclipse GUI and stores a backup copy of each modified file. The JUnit-action logger records the timestamp and the number of executed tests on each JUnit run. They then calculated the compliance value (the ratio of TDD + refactoring changes to all changes) by semi-automatically checking the difference between two versions of a file. Their results showed that experts comply with the TDD process more strictly than novices. And no participants consistently adhered to the TDD process.

Similarly, Latorre [Lat14] obtained development sequence data from two Eclipse plugins, one for transparently committing new revisions of files to a version control system each time students saved files locally, and one for collecting data on JUnit test invocations. The researcher then calculated the compliance value in the same way as Müller & Höfer [MH07]. And he found that the compliance

---

<sup>1</sup><http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>

value increased during the experiment. At the end of the experiment, all participants were able to successfully apply the test-driven approach.

Kou et al. [Kou10] created a system named Zorro, which was designed to automatically determine whether a developer complies with the TDD process. Zorro has three main steps: (1) it collects different kinds of developing events via IDE plugins; (2) it merges the events into high-level *episodes* (short-duration intervals of development); and (3) it classifies each episode into one of eight categories. During the experiments, they recorded the entire development step as a screencast for verification purpose. The results showed that on average Zorro can correctly classify more than 70% of episodes. Becker et al. [Bec15] built an automatic TDD compliance assessment framework called Besouro, which integrates important contributions from Zorro and other similar frameworks. Fucci et al. [Fuc14] used Besouro to classify development steps and used the same formula as Müller & Höfer [MH07] to calculate the compliance value. Their results did not show any significant relationship between compliance with the TDD process and its claimed effects.

Causevic et al. [Cau12b] asked students to commit modifications each time they created a test case. They referred to commit history to check students' development steps. Although it is easy to check commit history, it makes sense to do so only if students commit each time they finish writing a test or adding new code. Pančur & Ciglarič [PC11] examined compliance of students with the designated development approach by analyzing the data collected by an Eclipse plugin named *ProcessLog*. Then they subjectively evaluated the interleaving of coding and testing phases based on ProcessLog data and excluded the participants with weak TDD process compliance, which may introduce some bias into their results. Hilton et al. [Hil16] designed and implemented a tool named *TDDViz* that visualizes the TDD process based on code and test changes. Their experimental results showed that TDDViz can help developers better understand how they are adhering to the TDD process.

### 4.2.3 Challenges of Adherence to the TDD Process

Previous studies showed that it was not easy to adhere to the TDD process all the time [MH07; Bel15]. Beller et al. [Bel15] did a large-scale study with 416 software engineers over five months. They found that only ten developers strictly complied with the TDD process. The researchers proposed several reasons for the uneven adoption of TDD in practice: (1) developers might misunderstand the concept of TDD; (2) developers skip the testing step because they think their changes are so simple that they could not possibly fail; and (3) some code modifications (e.g., changes to configuration files) are not suitable for the test-driven approach.

Kou et al. [Kou10] documented the development process of the research tool *Zorro* over the course of eight weeks. The authors explicitly mentioned that they used the test-driven approach during the development [Kou07]. Results revealed that around 23% of episodes overall adhered to

the test-driven approach. Moreover, no episodes at all complied with the test-driven approach in three out of the eight weeks. The authors explained that since testing web interfaces requires a lot of additional work, the fraction of episodes adhering to the test-driven approach dropped significantly during that time.

To sum up, both industrial and academic professional developers are observed to apply test-driven techniques pragmatically rather than adhering to the TDD method all the time. Our course projects are based on a web application. Here too, it could be difficult to comply consistently with the test-driven approach throughout the whole project, which lasts more than one month. Consequently, we deemed students to be *pragmatically* in compliance with the test-driven approach if they spend about 20% of the time following the TDD process. Borle et al. [Bor17] introduced the concept of *TDD-like* to refer to development activity shown in the version-control system that are “emblematic of TDD but perhaps not perfectly.” In this chapter, we change the focus of the TDD-like concept from the version control system to steps recorded by IDE plugins.

### 4.3 Methodology

The purpose of our study is to help students learn the test-driven approach by using it to improve their contributions to OSS projects. Our research questions were:

- **RQ1:** Will students apply test-driven techniques pragmatically during course projects?
- **RQ2:** Does the test-driven approach help students write source code with fewer instances of common mistakes?
- **RQ3:** Does the test-driven approach help students write test cases with higher statement coverage?
- **RQ4:** Does the test-driven approach help students write test cases with better quality (mutation testing)?

#### 4.3.1 Tactics for Helping Students Complete Course Projects

A pre-survey of the students indicated that a large proportion of them have previous TDD experience and are willing to use the test-driven approach in course projects. However, few students had experience writing tests with RSpec framework. Moreover, Expertiza has more than 30,000 lines of source code, and it is a challenge for students to understand the existing design, involving hundreds of classes and dozens of database tables, become familiar with the testing framework and write tests. Our experience in previous semesters led us to conduct several tactics to help students become familiar with the test-driven approach and testing framework and understand what they need to do in course projects.

#### 4.3.1.1 Teaching and Tutorial Videos

Before starting the projects, all students studied the test-driven approach and RSpec testing framework for homework. In addition, we recorded a series of tutorial videos<sup>2</sup> to help students understand the syntax of the RSpec testing framework and to demonstrate how to refactor the Expertiza code-base while adhering to the test-driven approach. The last video ended by following links to several exemplary RSpec test files for the students' reference.

#### 4.3.1.2 Project Mentoring

It is not unique to course projects in this study. In fact, mentoring has been used in our course for several years. Each project has a mentor to help students to understand the project requirements and resolve confusion. Mentors can be the professor, the senior research engineer, Ph.D. students or teaching assistants. Students have to schedule weekly meetings with mentors (either face-to-face or online) to discuss the progress and issues. These meetings can make sure students understand what they need to do and keep them on the right track.

#### 4.3.1.3 Test Skeletons

Test skeletons are test methods that have signatures (specifications) but not test bodies. According to previous research [ES14; Hu18], tests generated by students could be shallow. We gave students test skeletons to direct their attention to conditions we wanted to test. For instance, we asked students to test the instance method `super_admin?` shown in Figure 4.1 (a). And we prepared two scenarios in test skeleton shown in Figure 4.1 (b) for students to complete. The first one says that when the role name of the current user is `Super-Administrator`, the method should return `true`; in the second scenario when the current user is not `Super-Administrator`, the method should return `false`.

Test skeletons are elaborated guidance. They allow students to figure out which kinds of tests are required. Students fill out the test skeletons with code like that shown in red in Figure 4.1 (c) and make the tests pass. It is worth noting that tests can be much more complicated than the example in Figure 4.1. Students are also responsible for writing tests for newly-created methods using the test-driven approach.

Before students started working on course projects, we cleaned up some existing shallow tests and wrote tests for all the projects. Then we removed the test bodies and committed the skeletons to GitHub. There are several reasons why we ourselves wrote tests. First, we were able to ensure that the code was clear before students began working on the projects. Second, test skeletons guided students to the kinds of tests we wanted them to write. Third, staff-authored tests can potentially be

---

<sup>2</sup><https://goo.gl/3dECtY>

```

1 def super_admin?
2   role.name == 'Super-Administrator'
3 end

```

(a)

```

1 describe '#super_admin?' do
2   it 'returns true if the role name of current user is Super-
    Administrator'
3
4   it 'returns false if the role name of current user is not Super-
    Administrator'
5 end

```

(b)

```

1 describe '#super_admin?' do
2   it 'returns true if the role name of current user is Super-
    Administrator' do
3     allow(user).to receive(:role).and_return(double(:role, name:
        'Super-Administrator'))
4     expect(user.super_admin?).to be true
5   end
6
7   it 'returns false if the role name of current user is not Super-
    Administrator' do
8     allow(user).to receive(:role).and_return(double(:role, name: 'Student'))
9     expect(user.super_admin?).to be false
10  end
11 end

```

(c)

Figure 4.1: An example of the method that needs to be tested (a), corresponding test skeleton (b) and completed tests (c)

Table 4.1: Twelve course projects

Project id and name <sup>a</sup>	Test skeleton
1. Refactor <code>assignment.rb</code>	<code>assignment_spec.rb</code>
2. Refactor <code>assignment_participant.rb</code>	<code>assignment_participant_spec.rb</code>
3. Refactor <code>user.rb</code>	<code>user_spec.rb</code>
4. Refactor <code>assignments_controller.rb</code>	<code>assignments_controller_spec.rb</code>
5. Refactor <code>questionnaires_controller.rb</code>	<code>questionnaires_controller_spec.rb</code>
6. Refactor <code>review_mapping_controller.rb</code>	<code>review_mapping_controller_spec.rb</code>
7. Refactor <code>assignment_form.rb</code>	<code>assignment_form_spec.rb</code>
8. Refactor <code>response.rb</code>	<code>response_spec.rb</code>
9. Refactor <code>team.rb</code>	<code>team_spec.rb</code>
10. Refactor <code>grades_controller.rb</code>	<code>grades_controller_spec.rb</code>
11. Refactor <code>response_controller.rb</code>	<code>response_controller_spec.rb</code>
12. Refactor <code>sign_up_sheet_controller.rb</code>	<code>sign_up_sheet_controller_spec.rb</code>

<sup>a</sup> Project 1-6 apply to the TDD group. Project 7-12 apply to the non-TDD group. Students are required to write unit tests (project 1-3 and 7-9) or integration tests (project 4-6 and 10-12).

used to check the correctness of students' modifications.

### 4.3.2 Experiment Design

We conducted our controlled study in the 2017 fall semester. We specified 12 course projects, listed in Table 4.1. Among them, six course projects were required to be done using the test-driven approach (TDD group; in each iteration, write tests first, then modify the code) and the remaining six ones were done using the traditional test-last approach (non-TDD group; modify code first, then write tests to validate modifications). We asked students to write unit tests for model files (in the Model-View-Controller framework) and integration tests for controller files. All tests were written in the RSpec testing framework. We offered test skeletons to both the TDD group and the non-TDD group. The reason why we chose these projects is that current statement coverage of each of these files was less than 50%, and some files did not have tests at all. So students were able to write tests first before refactoring the codebase. These selected files have many code “smells” and were rated “F” (worst) by Code Climate. Students were asked to optimize codebase and remove code smells. This helped to familiarize students with modifying existing code and understanding what complex coding environment looks like.

We evened out the workload among projects and limited the projects to the following three kinds of modifications: (1) splitting complex methods into smaller ones, (2) extracting duplicated code into new methods, and (3) using polymorphism to eliminate complicated switch statements. There are three reasons why we made this choice. First, it allows students to have more time to become familiar with RSpec syntax. Second, we are able to examine and compare the code quality of these



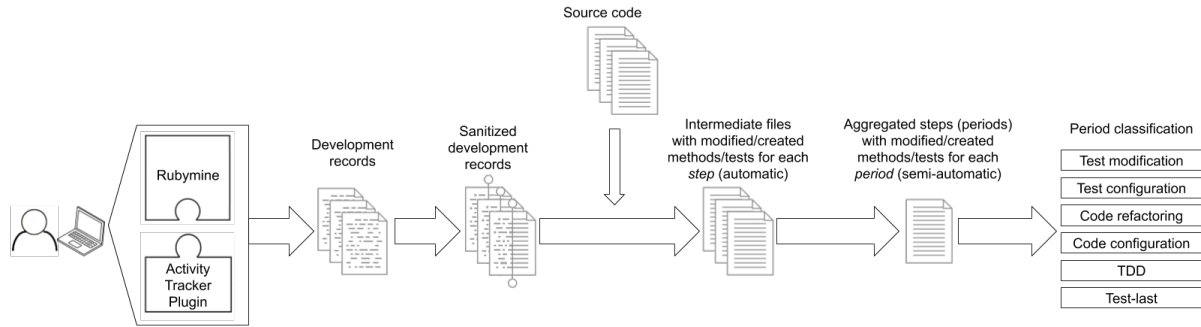


Figure 4.2: Data-processing pipeline

three modifications between the TDD group and non-TDD group. Third, these modifications are applicable to the test-driven approach. In other words, students are able to write failing tests first to describe the functionalities of expected methods, and then modify the code to make them pass.

#### 4.3.3 Examination of Compliance with the TDD Process

We need a way to inspect whether students comply with the test-driven approach and to what extent they follow it. As we mentioned in Related Work section, there are several ways to achieve this goal. However, many measurements are not suitable for our experiments, because our experiments are based on team projects rather than individual tasks. If we use the IDE plugins to commit changes to the version-control system each time students save files locally, there will be potentially many merge conflicts and it may disturb students during the development process. It is also difficult for us to manually analyze recorded videos because course projects last more than one month. Moreover, to our best knowledge, no automatic TDD process-recognition system (including Zorro [Kou10], Besouro [Bec15], TDDGuide [Mis08], ActivitySensor,<sup>3</sup> SEEKE [Obe09]) supports collecting development activity data from RubyMine (a Ruby on Rails IDE).

Figure 4.2 illustrates our data processing pipeline. We first used a RubyMine plugin named *Activity Tracker*<sup>4</sup> to record each student’s development steps and store them in csv files. Then we sanitized those development steps to make each step contain five data items: (1) timestamp, (2) current user, (3) keyboard events (character additions and deletions), (4) file opens in the text editor, and (5) current location of the cursor (line number and column number). Based on source code and sanitized development steps, we wrote a script to automatically generate intermediate files, which show the code after each development step by each student, as well as obtain the names of modified/created methods/tests for each development step. Then we semi-automatically aggregated the adjacent steps that modified/created the same methods/tests into *periods*. We also

<sup>3</sup><http://sens.e-informatyka.pl/projekty/activity-sensor/>

<sup>4</sup><https://plugins.jetbrains.com/plugin/8126-activity-tracker>

Table 4.2: Patterns followed in developing code and tests

Category	Pattern <sup>a</sup>
Test modification	Modifying the code in a test.
Test configuration	Managing testing fixtures or setting testing variables.
Code refactoring	Refactoring source code.
Code configuration	Setting up relationships between files ( <i>include</i> , <i>require</i> , etc.).
TDD <sup>b</sup>	Test modification <sub>1</sub> → Code refactoring <sub>1</sub> → ... [→ Test configuration <sub>i</sub> ] → Test modification <sub>m</sub> [→ Code configuration <sub>j</sub> ] → Code refactoring <sub>n</sub> [→ Test modification <sub>m+1</sub> ]
Test-last <sup>b</sup>	Code refactoring <sub>1</sub> → Test modification <sub>1</sub> → ... [→ Code configuration <sub>j</sub> ] → Code refactoring <sub>n</sub> [→ Test configuration <sub>i</sub> ] → Test modification <sub>m</sub> [→ Code refactoring <sub>n+1</sub> ]

<sup>a</sup> Subscript  $m$ ,  $n$ ,  $i$ ,  $j$  represent the order of different categories. The category in square brackets indicates that it is optional.

<sup>b</sup> Patterns for TDD or test-last category consist of atomic patterns from the other four categories, namely, test modification, test configuration, code refactoring, and code configuration. If a period uses only one of these atomic patterns, we classify the period into one of these four categories. Otherwise, we combine atomic patterns of these four categories, which modify/create one method or its corresponding tests, into the patterns of TDD or test-last category.

set a one-hour threshold to avoid the situation where one student stopped work on the project on one day and continued working on the same piece of code on the next day, so we didn't count all the time in between as time working on the project. Specifically, if the interval of two continuous steps was more than one hour, we considered them as two different periods and did not aggregate them. After that, we classified periods into six categories, namely, test modification, test configuration, code refactoring, code configuration, TDD, and test-last. Table 4.2 shows the pattern used to identify each category, which is informed by the heuristics presented in Kou [Kou10] and Fucci [Fuc17].

## 4.4 Data Collection

The subjects of our experiments are Computer Science and Computer Engineering masters students. Thirty-five students were involved in our study.

We collected both qualitative and quantitative data in this study. Qualitative data mainly came from surveys. We conducted two anonymous surveys, namely, a pre-survey, and a post-survey. We asked all 123 students enrolled in the course to respond to the pre-survey. The purpose of the pre-survey was to document the background of students, including industrial work experience, experience using the test-driven approach, and proficiency in the test-driven approach. For the post-survey, we only asked the 35 students who participated in our study to complete the survey. We tried to find out the most difficult part of course projects, measure the usefulness of test skeletons and the extent to which students followed the test-driven approach. We also asked students to write down their concerns, suggestions, and ideas about using the test-driven approach—which could be helpful for future course design.

We gathered quantitative data using two data sources: development steps as recorded by the

Table 4.3: Post-survey questions

Question statement	Question type
- Which is the most difficult part of your project?	Multiple choice
- Usefulness of the test skeletons.	5-point Likert scale
- (For the TDD group only) To what extent did you follow TDD process?	5-point Likert scale
- If you used VirtualBox image and RubyMine we offered to you, please write down your student id (for data collection purpose only).	Open ended
- Tell us any concerns, suggestions, ideas about using test-driven approach in course projects.	Open ended

Activity Tracker, and pull requests created by the students. We collected more than 157,000 development steps of 10 students. Then we sanitized and aggregated these steps into 2193 periods and classified each period into one category.

Each team submitted one pull request as the deliverable. We analyzed the code quality, including source-code modifications and newly added test cases. For the quality of the source code modifications, we checked the pull request merge status and counted the occurrence of 13 common mistakes (derived by analyzing 313 OSS-based course projects from 2012 to 2017 [Hu18]) in each project. For the quality of test cases, we analyzed 301 newly added tests, including 203 tests written by the TDD group and 98 tests generated by the non-TDD group. We used statement coverage and Mutation Score Indicator (MSI) to measure test thoroughness and fault-finding capability of tests.

It is worth noting that the three-member team that chose project 5 finished only two-thirds of their work because one team member became ill during the project period. Additionally, the team that chose project 7 modified the correct part of the source code but wrote integration tests for another file by mistake. So there is no team that wrote tests for project 7. Therefore, we excluded projects 5 and 7 from analyses related to testing.

## 4.5 Data Analysis

### 4.5.1 Survey Results

The response rate to the pre-survey was more than 91% (112 out of 123). We found that more than half of the students had more than one-year work experience in the industry, more than 40% of students have used test-driven approach before and only 6.3% of students were not willing to use the test-driven approach in course projects.

The response rate of the post-survey was almost 83% (29 out of 35). Table 4.3 lists post-survey questions. We deployed the post-survey to both TDD and non-TDD group. However, some survey questions were only applicable to the TDD group. More than half of students thought RSpec syntax

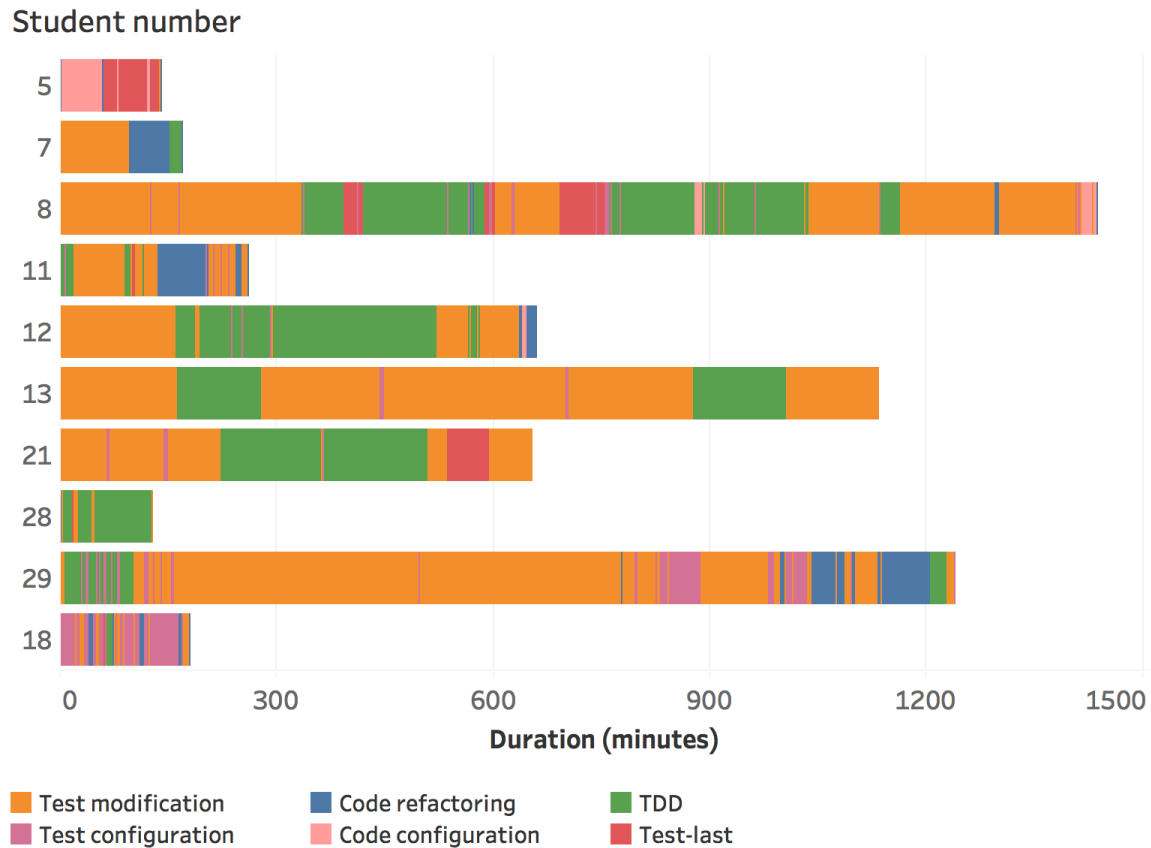


Figure 4.3: An overview of student development steps

was the most difficult part of course projects, followed by setting up Expertiza environment (13.8%), understanding object dependencies (10.3%), modifying the existing codebase (10.3%), and complying with the test-driven approach (10.3%). Further, 54% of the students in the TDD group claimed that they followed the test-driven approach strictly.

We tried several approaches to help students familiarize themselves with the test-driven approach and testing framework, and understand what they needed to do in course projects. Their comments indicate that both test skeletons and tutorial videos were quite helpful. Student 29 said that test skeletons were “*really of great help. It helped us understand which scenarios need to be handled with primary importance.*” Student 6 stated that the tutorial videos “*helped us understand the TDD approach and created a good starting point for the project.*” Also, student 19 mentioned that “*videos guided us on how to write well-structured test cases. They were very easy to follow.*”

Table 4.4: Merge status and common mistakes of 12 course projects

	TDD group						Non-TDD group					
Project id	1	2	3	4	5	6	7	8	9	10	11	12
Merge status <sup>a</sup>	P	M	P	M	R	R	P	M	M	R	P	P
Commenting <sup>b</sup>					Y					Y		Y
Shallow or missing tests							Y			Y		
Bad naming	Y				Y	Y				Y	Y	
Duplicated code	Y						Y					
Failing functionality			Y			Y					Y	Y
Messy pull requests										Y		
Hardcoding							Y					
Not following the existing design					Y							
Missing pull request/not forked												
Long methods												
Bad specification					Y							
Switch statements					Y	Y						
Limited modifications												

<sup>a</sup> There are three merge statuses, namely merged (M), rejected (R) and partially merged (P).

<sup>b</sup> The order of common mistakes is based on the frequency of these common mistakes. "Y" indicates that one project has a certain common mistake.

#### 4.5.2 Compliance with the TDD Process

Figure 4.3 illustrates the development steps followed by 10 students, including nine from the TDD group and one (student 18) from the non-TDD group. Each bar represents the aggregated development steps (periods) for one student. The length of the bar shows the duration of periods in minutes. In each bar, different colors indicate the categories of different periods.

On average, students in the TDD group spent 31% (median 22%) of their time adhering to the test-driven approach, while the student in the non-TDD group spent 6% (median 6%) of his time complying with the test-driven approach. Therefore, we consider that students in the TDD group adhered to the test-driven approach pragmatically during their course projects and those in the non-TDD group did not comply with the test-driven approach.

#### 4.5.3 Quality Analysis of Source-Code Changes

Table 4.4 shows merge status and common mistakes for 12 course projects. We defined three merge statuses, namely, merged (M), rejected (R) and partially merged (P). Partially merged status means that we merged only part of the student contribution, and refactored or removed the rest [Hu18]. In the TDD group, two out of six projects were merged into current codebase; another two projects

were partially merged. We rejected the other two projects. For the non-TDD group, we merged one-third of the projects, partially merged half of projects and rejected the last project.

We tallied the 13 common mistakes throughout 12 course projects. On average, the TDD group made 1.8 (median 1.5) common mistakes per project and the non-TDD group made 1.8 (median 2) common mistakes per project. We found that projects in both the TDD group and the non-TDD group exhibited commenting-related issues. Students either wrote meaningless comments, which could be avoided by adopting self-explanatory method/variable names or they commented out code instead of deleting it. Moreover, projects in both groups used bad method/variable names. Most of the time, students assigned these names for convenience without thinking about the understanding and maintenance cost for future contributors to the OSS project. Project 10 in the non-TDD group created a messy pull request, which included modifications coded by other teams. Students who worked on this project may have been unfamiliar with the operation of the Git version-control system. For three common mistakes mentioned above (commenting, bad naming, and messy pull request), adopting the test-driven approach or not would not seem to help much. One reason is that these common mistakes are more related to coding styles and version-control system familiarity instead of the functionalities.

What surprised us most is that we found projects 3 and 6 in the TDD group that had failing functionalities; that is to say, their modified code breaks some pre-existing features. After looking into these two projects, we found the root cause.

The goal of project 3 was to optimize the `User` class; and thus, students had to write unit tests. One requirement of this project was to utilize polymorphism to implement particular methods in child classes, such as `Instructor`. In the current codebase, all code is located in one method `get_user_list` shown in Figure 4.4 (a). Thus, the object that the `self` keyword refers to can be accessed by different `if` blocks. Figure 4.4 (b) displays the code as modified by students. Students extracted the code into a class method `self.get_user_list` located in the `Instructor` class. Then they called the newly-extracted method from the original `get_user_list` method located in Line 5 of Figure 4.4 (b).

There are two big issues with these modifications. First of all, students did not use polymorphism to refactor the code. What they did was simply to move the code to another file and call the new method from the original method. The correct modification is shown in Figure 4.4 (d). We can achieve polymorphism through inheritance by only declaring the method in the parent class and implementing the code in child classes. Secondly, besides the design issue, a runtime error is raised when the students' code is executed. Line 14 in Figure 4.4 (b) shows that the code tried to obtain the `id` attribute from the object that the `self` keyword refers to. However, the `self.get_user_list` method is a class method and the `self` keyword here represents the `Instructor` class itself, instead of an instance of the `Instructor` class. Since there is no way for us to access the `id` attribute without creating an instance of the class, a `NoMethodError` (undefined method `id`

```

1  # user.rb
2  class User < ActiveRecord::Base
3    def get_user_list
4      ...
5      if self.role.instructor?
6        ...
7        assgts=Assignment.where(instructor_id: self.id)
8        ...
9      end
10     ...
11   end
12 end

```

(a)

```

1  # user.rb
2  class User < ActiveRecord::Base
3    def get_user_list
4      ...
5      return Instructor.get_user_list if self.role.instructor?
6      ...
7    end
8  end
9
10 # instructor.rb
11 class Instructor < User
12   def self.get_user_list
13     ...
14     assgts=Assignment.where(instructor_id: self.id)
15     ...
16   end
17 end

```

(b)

```

1  # user_spec.rb
2  describe '#get_user_list' do
3    ...
4    allow(Instructor).to receive(:get_user_list).and_return([user1, user2
5      ])
6    ...
7  end

```

(c)

```

1  # user.rb
2  class User < ActiveRecord::Base
3    def get_user_list; end
4  end
5
6  # instructor.rb
7  class Instructor < User
8    def get_user_list
9      ...
10     assgts=Assignment.where(instructor_id: self.id)
11     ...
12   end
13 end

```

(d)

Figure 4.4: A piece of source code (a) and code after modified by students (b), the corresponding test written by students (c) and code refactored by course staff (d)

for Class) occurs when executing the code. Ideally, when extracting code blocks into new methods located in different files, we should pass any necessary variables as parameters of the new methods, or define different variables in the new methods. However, it seems that students simply copied the code from the parent class and pasted it into different subclasses, and did not try to execute the code.

One reason why students were not able to detect the failing functionality in a TDD cycle is that they only wrote unit tests for pre-existing methods and mocked behaviors of called methods. However, they did not write tests for newly-created methods before implementing them, which violates the project requirement—write tests for each newly-created method using the test-driven approach. Figure 4.4 (c) shows how students mocked the behavior of `Instructor.get_user_list` by returning two users as an array. Under this situation, the execution status of `Instructor.get_user_list` will not affect this test at all. In order to examine the correctness of the code and avoid this situation, we can always write unit tests for the newly-created method.

For project 6, we required students to optimize one controller file by splitting big methods into several smaller ones. Meanwhile, they needed to write integration tests. The concrete manifestation of failing functionality is that new methods created by students did not include all necessary parameters. Unlike unit tests, which require one or more test cases to be written for every method, when writing integration tests, students were only supposed to write tests for methods associated directly with views (public methods). For methods not invoked directly by user activity, also known as helper methods (private methods), students did not need to write specific tests. This is because private methods can be tested along with public methods that call them. They should *not* write redundant tests for helper methods. After investigating the pull request of project 6, we noticed that although students wrote integration tests for the original big methods, they mocked behaviors of separate smaller methods (helper methods) rather than invoking these helper methods directly during integration tests. In this case, integration tests did not test the functionalities of separate helper methods (which had been mocked rather than invoked). This is why integration tests passed but the separate helper methods had failing functionalities.

Based on merge status and the number of common mistakes, we did not find major differences in the quality of source code modifications between both groups.

#### 4.5.4 Test Quality Analysis

Table 4.5 shows the overview of pre-existing tests, staff-authored tests, and student-authored tests. Projects 1–6 are those completed by the TDD groups and projects 7–12 were completed by the non-TDD group. We noticed that four out of six teams in the TDD group (but no teams in the non-TDD group) wrote either more test cases or achieved higher statement coverage than the teaching staff. This indicated that students in the TDD group wrote tests for newly created methods, besides tests



Table 4.5: Overview of pre-existing (p)<sup>a</sup>, staff-authored (t)<sup>b</sup> and student-authored tests (s)<sup>c</sup>

	TDD group						non-TDD group					
Project id	1	2	3	4	5	6	7	8	9	10	11	12
Number of tests (p)	23	8	11	0	0	0	4	0	2	3	9	20
Number of tests (t)	55	30	58	22	34	39	28	15	28	11	23	27
Number of tests (s)	57	38	62	22	25	39	–	15	28	5	23	27
Statement coverage (p) (%)	41.2	32.0	33.3	0.0	0.0	0.0	39.3	0.0	22.7	13.5	30.6	38.3
Statement coverage (t) (%)	82.3	90.3	83.3	87.7	84.7	71.8	85.1	89.1	93.0	83.8	90.6	89.9
Statement coverage (s) (%)	77.6	86.2	82.8	89.1	46.8	66.0	–	66.9	68.0	52.3	79.0	86.8
MSI (p) (%)	0.8	1.4	9.0	1.2	0.9	0.6	13.2	0.6	15.9	0.8	51.8	0.6
MSI (t) (%)	92.0	5.0	68.1	82.8	56.9	59.0	69.4	86.4	70.1	80.4	84.5	81.7
MSI (s) (%)	82.9	5.1	60.6	81.4	25.0	41.6	–	74.8	73.1	86.9	76.1	75.0

<sup>a</sup> The data is based on pre-existing tests only.<sup>b</sup> The data is based on pre-existing and staff-authored tests.<sup>c</sup> The data is based on pre-existing and student-authored tests.

mentioned in test skeletons, which is exactly what we expected them to do. On average, the TDD group wrote 2.8 (median 2.0) more tests than the teaching staff and the non-TDD group wrote 1.2 (median 0.0) fewer tests than teaching staff. This indicates that the TDD group tended to write more tests.

Descriptive statistics showed that tests written by the TDD group had on average 91% (median 100%) statement coverage per method, and tests generated by the non-TDD group had on average 79% (median 92%) statement coverage per method.

We also examined the statistical significance of statement coverage between tests written by the TDD group and tests generated by the non-TDD group. The result of one-sided Wilcoxon rank-sum test ( $p = 0.00044 < 0.05$ ) revealed that the TDD group tended to write tests that achieved significantly higher statement coverage than the non-TDD group.

We performed mutation testing and calculated the MSI to examine the quality of the tests. For instance, an MSI of 92% means that 92% of all generated mutants were recognized as incorrect (e.g., they generated kills, timeouts or fatal errors).<sup>5</sup> On average, tests written by the TDD group had 74% (median 78%) MSI per method, and tests created by the non-TDD group had 65% (median 68%) MSI per method.

We tested MSI as well as statement coverage for statistical significance. The result of one-sided Wilcoxon rank-sum test ( $p = 0.058 > 0.05$ ) showed that the MSI difference between the TDD group and the non-TDD group was not significant. Thus, statistical results do not provide a basis for concluding that tests written by the TDD group are significantly different in quality than those written by the non-TDD group.

During code review, we noticed several issues in student-authored tests. The first issue is shal-

<sup>5</sup><https://github.com/humbug/humbug#the-metrics>

low tests (which occurred in projects 7 and 10). Although we offered test skeletons and created tutorial videos for students, some of them still wrote incomplete or shallow tests by either simply commenting out some failing test code or writing test code without expectations. Another scenario is that students only expected one object to receive a certain method without return values. For instance, students wrote the code `expect(user).to receive(:calculate_scores)` to make sure this user receives the `calculate_scores` method. However, no further expectations were included in this test cases, which means no matter what the return value of this method is, this test will pass as long as this user called the `calculate_scores` method. Ideally, test code mentioned above should test the return value, like `expect(user.calculate_scores).to eq(100)`. All scenarios mentioned above will make tests always pass; however, these tests will be less useful than others.

Another issue we found relates to mock behaviors in tests. Many students preferred to use the `any_args` keyword, which means all kinds of arguments are permissible. However, too many wild cards will make tests insensitive to code modifications. Suppose we want to test this code: `user=User.find_by(name:'name', age:25)`. We could write a mock method `find_by` by `allow(User).to receive(:find_by).with(any_args).and_return(user)`. If someone accidentally changed the code to `user=User.find_by(age:'name', age:25)`, the test code above will still be valid since `any_args` represents all kinds of arguments. A safer way to write this mock behavior is `allow(User).to receive(:find_by).with(name:'name', age:25).and_return(user)`, which makes the test more sensitive to argument modifications.

Based on our analysis, we observed that the TDD group tended to write more tests. Their test quality was almost the same as the non-TDD group's, but the statement coverage was significantly higher.

## 4.6 Discussion

For RQ1, we examined how students applied test-driven techniques pragmatically during their course projects. The analysis of student development steps shows that students in the TDD group spent on average 31% of their time complying with the test-driven process. Based on our definition of *pragmatically*, we claim that students in the TDD group adhered to the test-driven approach pragmatically during course projects.

To answer RQ2, we were not able to conclude that the test-driven approach helped students write source code with fewer common mistakes. According to merge status and common mistakes in 12 course projects, we did not find much difference in the quality of source code modifications between the TDD group and the non-TDD group. Also, the test-driven approach did not help avoid many types of common mistakes, such as commenting issues, bad naming, and messy pull requests.

For RQ3, we concluded that the test-driven approach helped students write tests with higher

Table 4.6: Average percentage of commits per team per week

Weeks	1 <sup>a</sup>	2	3	4	5
TDD group	0.6%	16.4%	27.4%	31.3%	24.3%
non-TDD group	0.0%	0.8%	30.4%	49.1%	19.7%

<sup>a</sup> Week 1 is for the teams to discuss the project details with teaching staff. Therefore, there were fewer commits than later weeks.

statement coverage. According to the statistical analysis, we found that the TDD group tended to write tests that achieved significantly higher statement coverage than the non-TDD group.

To answer RQ4, we did not conclude that the test-driven approach helped students write tests with better quality. Our statistical analysis failed to find a significant difference in test quality between the TDD group and the non-TDD group.

There are discrepancies between statement coverage and MSI on both staff-authored tests and student-authored tests. Most of the time, MSI is lower than the statement coverage, which is predictable. It is because our test skeletons aim to help students write tests, instead of covering all corner cases and achieving a very high branch coverage. For project 2, the overall MSI is much lower than statement coverage. That is because mutation testing created large numbers of mutants based on several methods that were not covered by test skeletons. Hence both staff-authored tests and student-authored tests were not able to detect these mutants.

We further mined the pull requests submitted by the TDD group and the non-TDD group. We found that on average the TDD group started the project one week earlier than the non-TDD group. According to the Table 4.6, the average percentage of weekly commits indicates that the test-driven approach can induce the team to even out the weekly workload.

We also conducted a larger-scale controlled study with 48 students. Every student team was required to implement new features for Expertiza. Many of them involved testing web interfaces. Hence, we prepared detailed user stories and asked students to write high-level feature tests (acceptance tests). This experiment was more like asking students to use the Behavior-Driven Development (BDD) approach. According to our survey, only about one-fifth of students in this experiment claimed that they followed the test-driven approach strictly. Students mentioned that “there is a steep learning curve to figure out how to use the feature tests” and “feature test is totally different from controller test.” Due to the difficulty of writing feature tests, we recommend that instructors who introduce test-driven approach in class or course projects control the granularity of testing and pay more attention to unit tests and integration tests.

## 4.7 Threats to Validity

### 4.7.1 Internal Validity

The main threat to internal validity is that students were inexperienced developers. Even when we provided several tactics to help them learn the test-driven approach, they might not have sufficient understanding of TDD. Moreover, students chose to join the TDD group or the non-TDD group on their own volition. It is possible that students with more experience in the test-driven approach joined the TDD group. Although the pre-survey asked about the students' previous experience, we were not able to balance both groups because the survey was anonymous. Besides, we required students to utilize the test-driven approach to optimize current code and fix some glitches, instead of implementing new features. This may be the reason that there were no major differences in the quality of source code modification. We offered test skeletons to students to facilitate writing tests. However, test skeletons may have induced students to write more tests, especially students in the non-TDD group.

In our post-survey, we did not ask students from the non-TDD group to answer the survey question, To what extent did you follow the test-driven process. Furthermore, our conclusion that students in the non-TDD group did not comply with the test-driven approach was based on the development steps of only one student. Thus, we cannot be sure that students from the non-TDD group did not inadvertently apply the test-driven approach during the course project.

### 4.7.2 External Validity

Results based on 35 masters students enrolled on our course may not be generalizable to all masters students. In addition, many other factors may affect student development steps, such as the complexity of a particular method, the deadlines of the course project or the workload of other courses.

### 4.7.3 Construct Validity

We utilized keyboard activity recorded by the Activity Tracker to measure to what extent students followed the test-driven approach. We did not collect the test execution data during the course project. Therefore, the pattern used to identify each category is primarily based on the sequencing of different periods, which may not 100% reflect the actual development steps. Also, keyboard activity included *copy* and *paste* behavior, but data on how many lines were copied or pasted was not recorded. Due to this fact, the code in intermediate files may be off by several lines. We manually checked the position of the code and tried to correct it if necessary. We were not able to fix all instances because information is lacking. This may have led to inaccurate category identification.

and period classification.

## **4.8 Conclusions**

We introduced several tactics to help students learn the test-driven approach and studied whether this approach is able to improve student contributions to OSS projects. We found that students in the TDD group were able to apply test-driven techniques pragmatically for the entire multi-week project period. Although we didn't find major differences in the quality of source code modifications and newly created tests between two groups, the TDD group wrote more tests and achieved significantly higher statement coverage—12% more statement coverage, on average.

# AN INTERNET BOTS APPROACH TO IMPROVING FEEDBACK ON GITHUB PULL REQUESTS

## 5.1 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 [Ger17; Gul14; Fal14; Sin13; Dou05] 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 [Keu16]. 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 OODD 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 integrated Code Climate and Travis CI with GitHub. Although these tools can help student modifications to 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 chapter, 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 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<sup>1</sup> 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.

## 5.2 Related Work

Previous research has explored different approaches to providing automated and timely feedback to students programming assignments or projects. Keuning et al. [Keu16] summarize three different techniques to achieve this goal. The first one is to adopt domain-specific techniques, such as

---

<sup>1</sup>There were 461 course projects related to Expertiza from the 2012 fall semester to the 2018 fall semester, including 63 projects that did not submit pull requests and two teams that used the same pull requests for both the initial and final project (viz., <https://github.com/expertiza/expertiza/pull/411>, <https://github.com/expertiza/expertiza/pull/1078>). Therefore, there were 396 unique pull requests from the 2012 fall semester to the 2018 fall semester.

static code analysis, dynamic code analysis using automated testing, program transformations, and intention-based diagnosis. Gulwani et al. [Gul14] 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 et al. [Sin13] 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 [SLB08] 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 et al. [Ger12] 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 & Singh [BS16] 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 et al. [Iha10] 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 et al. [Keu16] 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.



## 5.3 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?

### 5.3.1 Internet Bots

**Danger Bot:** We created the Danger Bot based on a Ruby gem named *Danger*.<sup>2</sup> 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 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 Table 5.1. The code for identifying each guideline has been committed to the Expertiza GitHub repository.<sup>3</sup> Figure 5.1 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*.<sup>4</sup> 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 Figure 5.2. 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.

---

<sup>2</sup><https://github.com/danger/danger>

<sup>3</sup><https://github.com/expertiza/expertiza/blob/master/Dangerfile>

<sup>4</sup><https://github.com/bluzi/travis-buddy>

Table 5.1: System-specific guidelines

<b>Id</b>	<b>System-specific guideline</b>	<b>Category</b>
1	Your pull request should not be too big (more than 500 LoC).	Big pull request
2	Your pull request should not be too small (less than 50 LoC).	Small pull request
3	Your pull request should not touch too many files (more than 30 files).	Many file changes
4	Your pull request should not have too many duplicated commit messages.	Duplicated commit messages
5	Your pull request is "work in progress" and it will not be merged.	WIP pull requests
6	Your pull request should not contain "Todo" or "Fixme" keyword.	Todo/fixme tasks
7	Your pull request should not include temp, tmp, cache file.	Temp files
8	Your pull request should avoid using global variables and/or class variables.	Global/class variables
9	Your pull request should avoid keeping debugging code.	Debugging code
10	You should write tests after making changes to the application.	Missing tests
11	Your pull request should not include skipped/pending/focused test cases.	Skipped/pending/focused tests
12	Unit tests and integration tests should avoid using "create" keyword.	Keyword in tests
13	RSpec tests should avoid using "should" keyword.	Keyword in tests
14	Your RSpec testing files do not need to require helper files (e.g., rails_helper.rb, spec_helper.rb).	Require helper files in tests
15	You should avoid committing text files for RSpec tests.	Commit text files for tests
16	Your pull request should not change or add *.md files unless you have a good reason.	*.md file changes
17	Your pull request should not change DB schema unless there is new DB migrations.	DB schema changes
18	Your pull request should not modify *.yml or *.yml.example file.	Config file changes
19	Your pull request should not modify test-related helper files.	Config file changes
20	Your pull request should not modify Gemfile, Gemfile.lock.	Config file changes
21	You should not change .bowerrc.	Config file changes
22	You should not change .gitignore.	Config file changes
23	You should not change .mention-bot.	Config file changes
24	You should not change .rspec.	Config file changes
25	You should not change Capfile.	Config file changes
26	You should not change Dangerfile.	Config file changes
27	You should not change Guardfile.	Config file changes
28	You should not change LICENSE.	Config file changes
29	You should not change Procfile.	Config file changes
30	You should not change Rakefile.	Config file changes
31	You should not change bower.json.	Config file changes
32	You should not change config.ru.	Config file changes
33	You should not change setup.sh.	Config file changes
34	You should not modify vendor folder.	Config file changes
35	You should not modify /spec/factories/ folder.	Config file changes
36	You should not commit .vscode folder to your pull request.	Config file changes
37-41	RSpec tests should avoid shallow tests.	Shallow tests



**danger-bot** commented 18 days ago • edited ▾

Contributor + 😊 ...

1 Message

⋮

Thanks for the pull request, and welcome! 🎉 The Expertiza team is excited to review your changes, and you should hear from us soon.

This repository is being automatically checked for code-quality issues using `Code Climate`. You can see results for this analysis in the PR status below. Newly introduced issues should be fixed before a pull request is considered ready to review.

Also, please spend some time looking at the instructions at the top of your course project writeup.

If you have any questions, please send email to [expertiza-support@lists.ncsu.edu](mailto:expertiza-support@lists.ncsu.edu).

1 Warning

⚠️

In your tests, there are many expectations of elements on pages, which is good. To avoid `shallow tests` – tests concentrating on irrelevant, unlikely-to-fail conditions – please write more expectations to validate other things, such as database records, dynamically generated contents.

✅

There are code changes, but no corresponding tests. Please include tests if this PR introduces any modifications in behavior.

✅

There are one or more skipped/pending test cases in your pull request. Please fix them so they run.

Figure 5.1: A pull-request comment created by the Danger Bot



**expertiza-travis...** commented 26 days ago • edited ▾

+ 😊 ...

### Travis tests have failed

Hey @Winbobob,  
Please read the following log in order to understand the failure reason.  
It'll be awesome if you fix what's wrong and commit the changes.

**Ruby: 2.2.7**

```
► export DISPLAY=:99.0 && RUBYOPT=W0 bundle exec parallel_rspec spec/$TESTFOLDER 2> /dev/null
```

Figure 5.2: A pull-request comment created by the Travis CI Bot

**Code Climate Bot:** The Code Climate Bot can insert pull-request comments (shown in Figure 5.3a) and inline comments (shown in Figure 5.3b) 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.

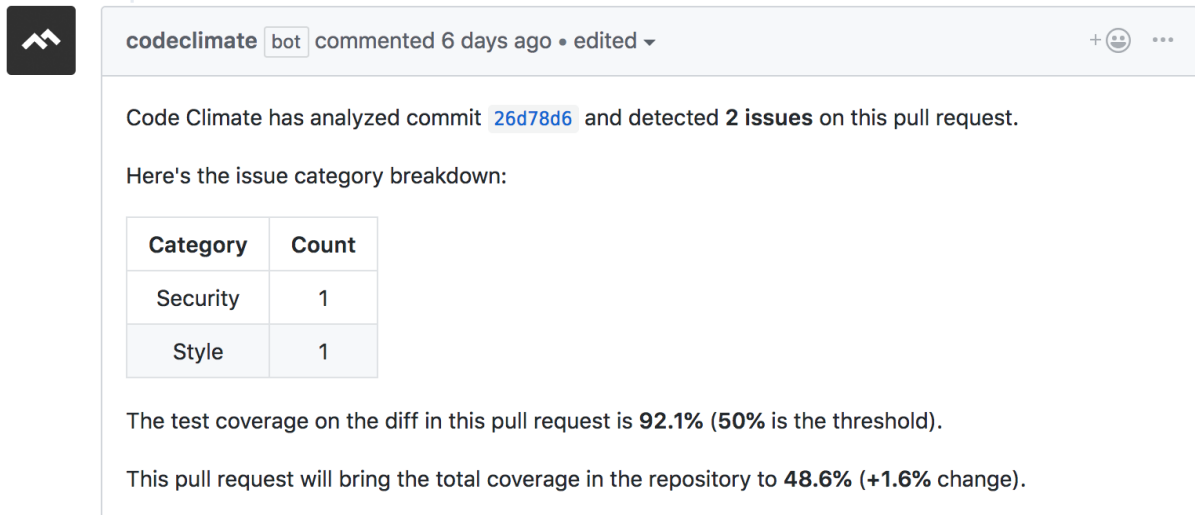
### 5.3.2 Bot Behavior Adjustment

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 RSpec 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



codeclimate bot commented 6 days ago • edited ▾

Code Climate has analyzed commit [26d78d6](#) and detected **2 issues** on this pull request.

Here's the issue category breakdown:

Category	Count
Security	1
Style	1

The test coverage on the diff in this pull request is **92.1%** (50% is the threshold).

This pull request will bring the total coverage in the repository to **48.6%** (+1.6% change).

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



app/models/assignment\_form.rb

```

200 +
201 +   queue = Sidekiq::ScheduledSet.new
202 +   queue.each do |job|
203 +     assignmentId = job.args.first

```

codeclimate bot 17 days ago

Use snake\_case for variable names.

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

Figure 5.3: Comments created by the Code Climate Bot

Table 5.2: Survey questions

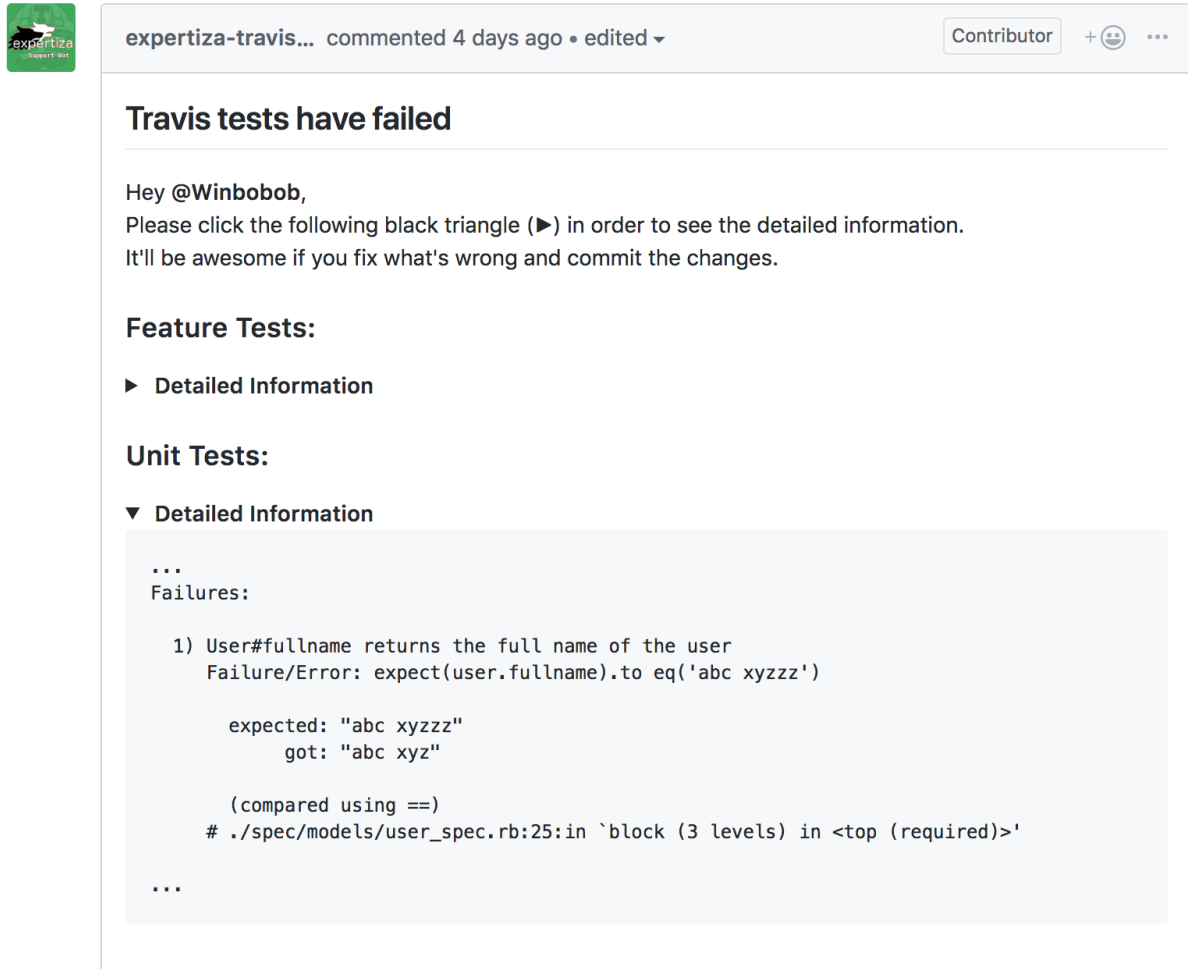
Survey Type	Question Statement	Question Type
Pre-survey	- How long have you spent working in industry (including internships)?	Multiple choice
	- Have you used internet bots before, e.g., chat bots?	Multiple choice
	- Have you developed internet bots before, e.g., Slack bots?	Multiple choice
Post-survey	- 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

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 Figure 5.4a, 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 Figure 5.4b). The upgraded Travis CI Bot can provide more timely feedback and help students focus on failed tests.

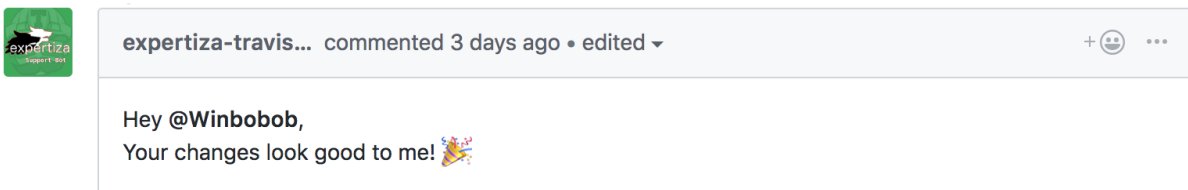
**Eliminating Overwhelming Inline Comments Created by the Code Climate Bot:** We conducted a poll on Piazza and asked students whether the comments generated by bots were overwhelming. We received votes from one-third of the students. The results indicated that almost 66% of students considered that 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<sup>5</sup> 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.

<sup>5</sup>The average number of pull request comments in the initial course project.



(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.

Figure 5.4: Comment created by the new Travis CI Bot

## 5.4 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 5.2 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. 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.<sup>6</sup> 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 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).<sup>7</sup> 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

---

<sup>6</sup><https://github.blog/changelog/2018-05-24-comment-edit-history/>

<sup>7</sup>We cannot obtain the test execution results for pull requests when they had merge conflicts.



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<sup>8</sup> 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).

## 5.5 Data Analysis

### 5.5.1 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*

---

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

*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.

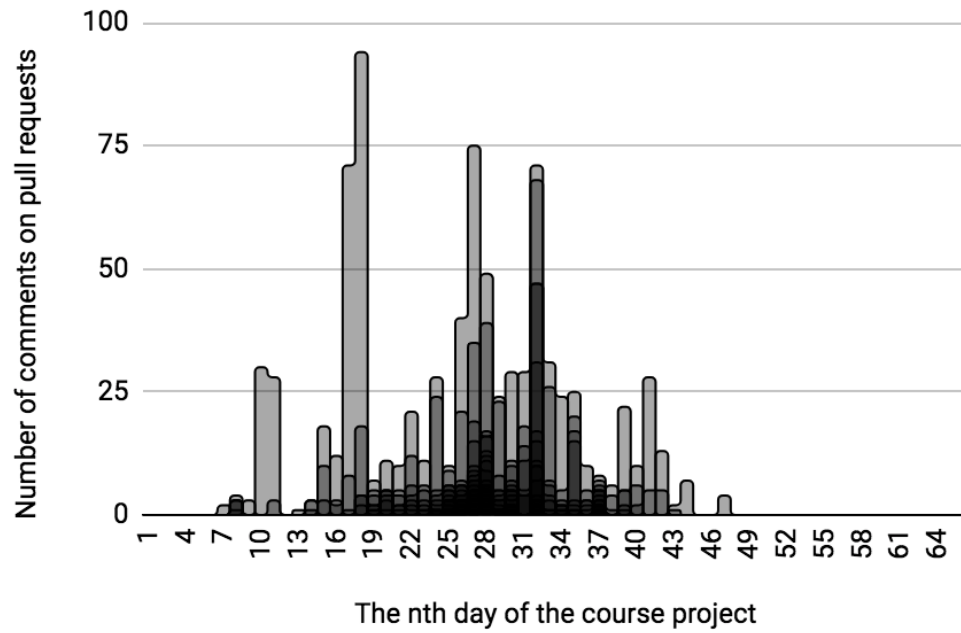
### 5.5.2 Overview of Pull Request Comments

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. Figure 5.5a exhibits the comments on 57 pull requests in the 2018 fall semester. Figure 5.5b displays the comments for the remaining 339 pull requests from semesters before 2018 fall. In both figures, the  $x$ -axes represent the  $n$ th day of the course projects, and the  $y$ -axes show the number of comments on pull requests. Each bar represents the comments 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.

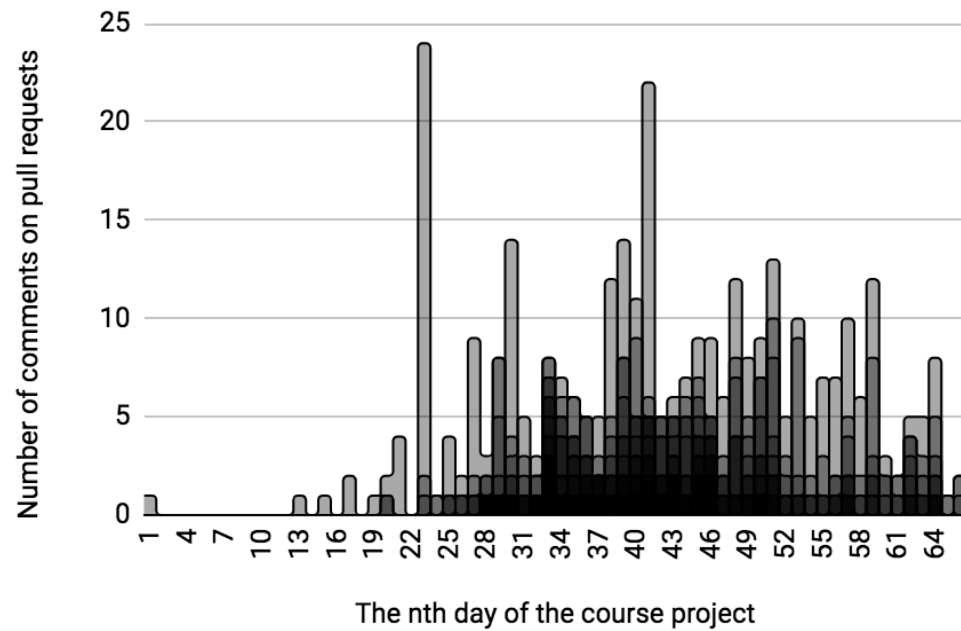
We analyzed when the last comment was made on each pull request in the 2018 fall semester and semesters before 2018 fall. On average, the last comment was created on day 34 (median 35) in the 2018 fall semester and on day 45 (median 44) in semesters before 2018 fall. We performed tie correction and applied 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.

### 5.5.3 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. Figure 5.6a and Figure 5.6b show re-tallied comments generated by bots in the 2018 fall semester and re-tallied comments created by teaching staff in

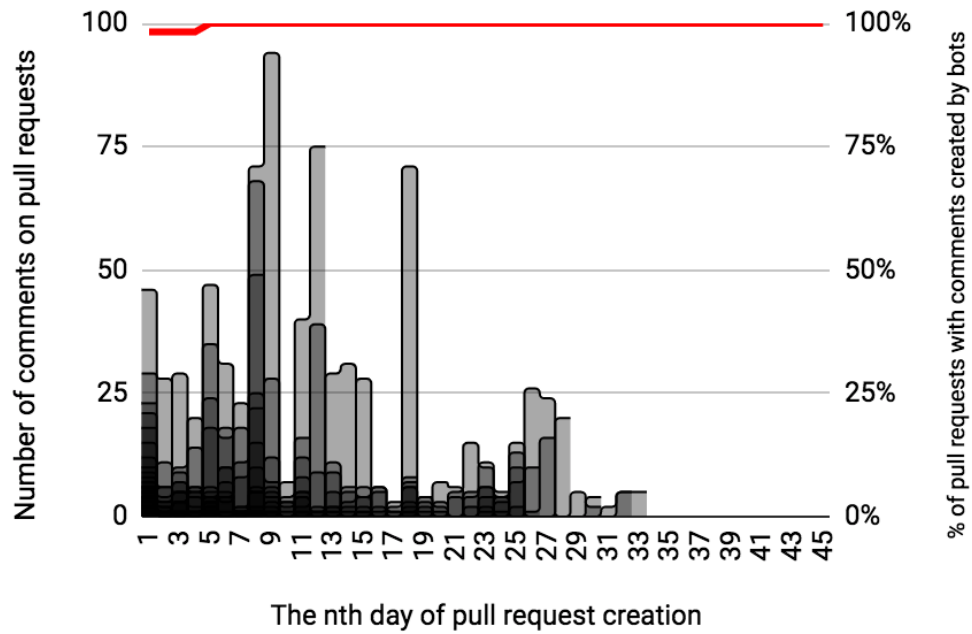


(a) Timing of comments on course projects in fall 2018

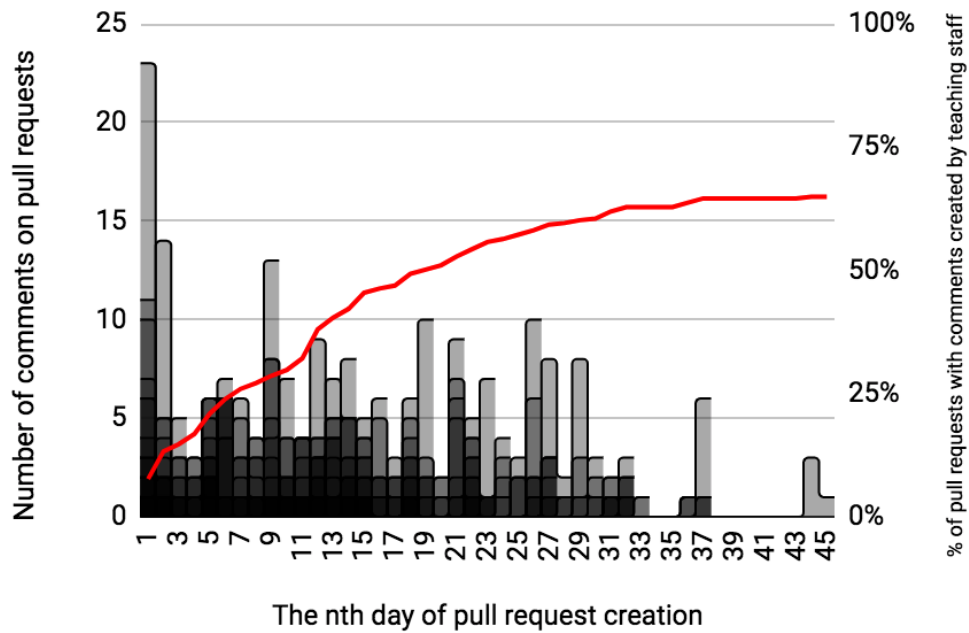


(b) Timing of comments on course projects in semesters before 2018 fall

Figure 5.5: 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

Figure 5.6: Comments for all course projects (Day 1 is the first day of pull-request creation. The last comment was created on day 66 of one course project, which is the 45th day since that pull request was created.)

Table 5.3: Contingency table for Travis CI results

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

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 Figure 5.6a and Figure 5.6b, we then calculated a new metric—the *percentage of pull requests with comments*. This metric is shown as red lines in Figure 5.6. We found that bots created comments on 98%<sup>9</sup> 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 Figure 5.6b, 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.

#### 5.5.4 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. The top 3 guideline violations that students resolved are guidelines 2, 9, and 10 in Table 5.1.

<sup>9</sup>The reason why bots did not create comments on all pull requests on the first day of pull-request creation was that there was a team in the 2018 fall semester that continued to work on a big project across the initial and the final course projects. They did not resolve the pull request merge conflicts until several days after they created the pull request. Hence, the bots could not analyze their intermediate pull requests (with merge conflicts) in the meanwhile.

### 5.5.5 Automated Test Results

We constructed a contingency table (Table 5.3) 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.

### 5.5.6 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.

## 5.6 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. Although the Danger Bot has not only checked modified files but also analyzed added/renamed/deleted files, 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`

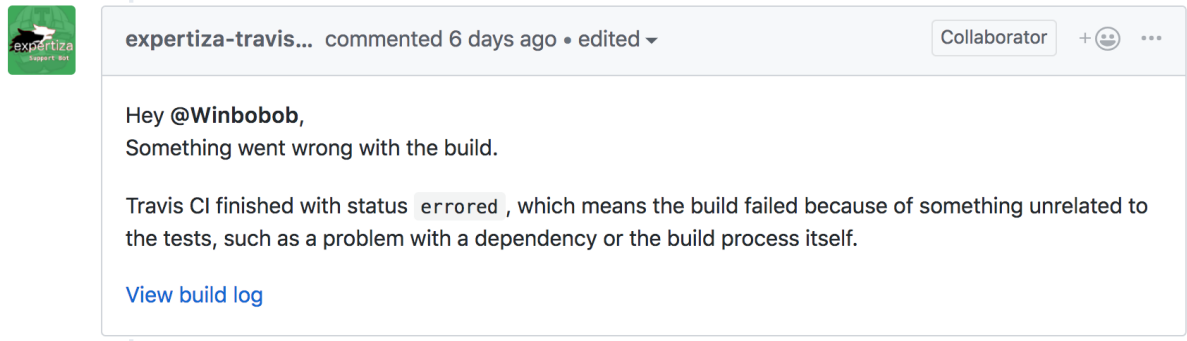


Figure 5.7: A pull-request comment created by the Travis CI Bot shows that test execution cannot be completed due to an error.

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, the comment of the Travis CI Bot is shown in Figure 5.7. Compared with Figure 5.4a, Figure 5.7 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 chapter. They highlight different, related, perspectives on course projects. Figure 5.8 depicts the relationship between them. The left column shows the types of 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. Figure 5.8 indicates that

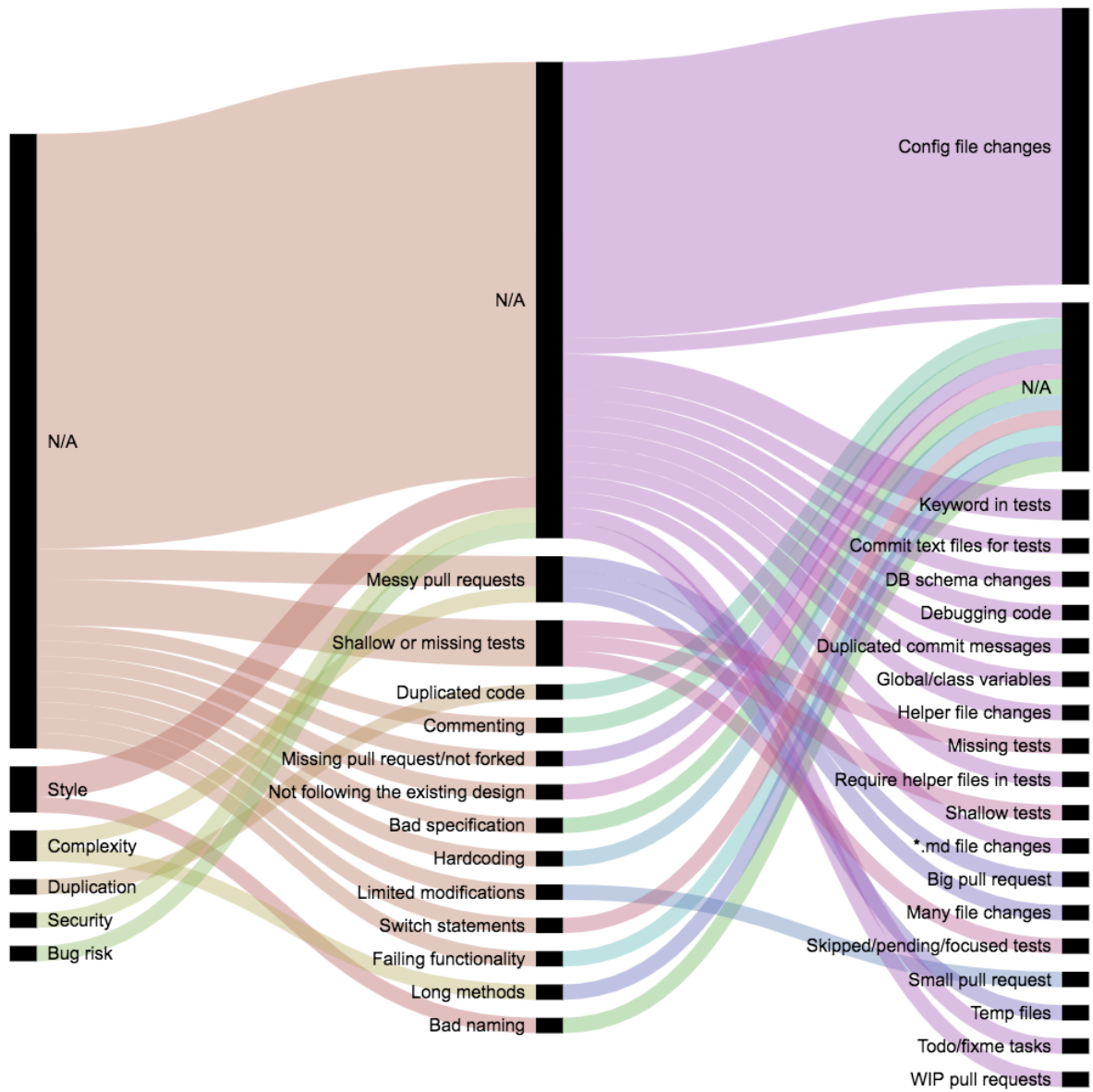


Figure 5.8: Relationship between code smells (left column), common mistakes (middle), and system-specific guideline categories (right). The thickness of each line corresponds to the number of the problem types.



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 OSS projects.

We plan to program bots to automatically collect all three metrics. The Code Climate Bot has been able to automatically detect five types of code smells and the Danger Bot has been able to automatically identify more than 40 system-specific guideline violations. Besides, the existing bots have been able to automatically detect some common mistakes, such as bad naming, long methods, duplicated code, as shown in Figure 5.8. We tried to make bots automatically detect the remaining common mistakes. Here we focused on shallow or missing tests. This is because according to Figure 3.5, 203 out of 335 rejected or partially merged course projects had problems of shallow or missing tests, which is one of the most frequent mistakes. We implemented seven system-specific guidelines to automatically detect the shallow-or-missing-test mistake based on the criteria in Table 3.1. The first guideline (guideline 10 in Table 5.1)<sup>10</sup> checks whether students wrote tests for the newly-added/modified code. The second guideline (guideline 11)<sup>11</sup> examines whether there were skipped/pending/focused tests in student contributions. The rest five guidelines (guidelines 37-41)<sup>12</sup> encompass the remaining criteria for shallow tests.

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.

## 5.7 Threats to Validity

### 5.7.1 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.

---

<sup>10</sup><https://github.com/expertiza/expertiza/blob/master/Dangerfile#L201>

<sup>11</sup><https://github.com/expertiza/expertiza/blob/master/Dangerfile#L214>

<sup>12</sup><https://github.com/expertiza/expertiza/blob/master/Dangerfile#L488>

### 5.7.2 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.

### 5.7.3 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: running tests in parallel can inflate the number of comments created by bots;<sup>13</sup> 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.

## 5.8 Conclusions

In this chapter, 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.

---

<sup>13</sup>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.

## CHAPTER

# 6

## SUMMARY AND FUTURE WORK

### 6.1 Summary

This report summarizes our work on helping students make better contributions to OSS projects. We found some quality problems in student contributions during our long-term maintenance of a student-authored open-source software application. Then we implemented several mechanisms, including test-driven development and Internet bots, to help students improve their contributions to OSS projects.

Chapter 1 introduces a student-authored open-source software application named Expertiza, and tells how Expertiza is integrated into our software engineering course and became the main code base for the course. In the history of the project, hundreds of students have contributed code and helped the software undergo several major updates. Student contributions also greatly improved the performance and the UI of Expertiza.

Chapter 2 shares our long-term experience on how to maintain Expertiza with limited resources. We discuss what challenges instructors would meet when they try to maintain and run a student-authored software application from four perspectives: code quality, code review and deployment process, infrastructure, and human resources. We also discuss what approaches instructors could use to overcome these challenges for each perspective.

Chapter 3 shares our experience in analyzing 313 OSS-based course projects from 2012 to 2017. We manually checked these course projects and identified 13 mistakes that frequently occur in

student contributions. These 13 common mistakes can be made into a checklist that helps code reviewers to check the quality of contributions. We then give four suggestions to help students reduce the frequency of common mistakes and improve the quality of their OSS pull requests. We hope these suggestions will also guide other instructors when using OSS projects as a code base for a software-engineering course.

Chapter 4 presents our work in conducting a quasi-experimental controlled study, which lasted more than one month, to test the performance of the test-driven approach on OSS-based course projects. We found that students were able to apply test-driven techniques pragmatically throughout the entire project. There were no major differences in the quality of source-code modifications and newly added tests between the TDD group and the non-TDD group; however, the TDD group wrote more tests and achieved significantly higher test coverage—12% more statement coverage, on average.

Chapter 5 exhibits our work on using three Internet bots to improve feedback on GitHub pull requests. These bots supplement the existing static code analyzer and continuous integration service by helping detect more than 40 system-specific guideline violations, and explicitly displaying test execution results and code-smell advice on pull-request pages. Survey results showed that more than 84% of students thought bots helped them to make better contributions. The Danger Bot is associated with a significant reduction of guideline violations, by 39%, and the Code Climate Bot is associated with a significant 60% decrease of code smells. But we did not find that the Travis CI Bot helped student contributions pass automated tests.

## 6.2 Future Work

**Extension of the common-mistake list and a scoring mechanism for common mistakes.** We have summarized 13 common mistakes. However, we have not covered several kinds of mistakes, such as security. In the future, we can include a wider variety of mistakes. Currently, we treat all common mistakes equally. However, we have noticed that some mistakes (e.g., not following the existing design) can have a greater impact on code base than others (e.g., commenting). We can design a scoring mechanism to determine which kind of mistake has more impact on the system and offer code reviewers more accurate feedback.

**Auto-generated test skeletons from source code.** According to our survey results in Chapter 4, students felt that test skeletons were quite helpful. We also mentioned that we have to manually create test skeletons to help students write complicated tests. This was a very time-consuming task. Therefore, we plan to use sequence-to-sequence learning techniques [Sut14] to automatically generate test skeletons from source code. The input of the sequence-to-sequence model will be a method; the output of the model will be the corresponding test skeletons. In this way, we can continue to provide test skeletons to help students write tests without too much effort.

**Further enhancement of bots.** Students mentioned that bots only displayed the system-specific guideline violations without locating corresponding files and methods. We will enhance bots to include this feature and add some good and bad examples as part of feedback provided by bots to help students learn best coding practices. We plan to make bots run locally to dissuade students from making dummy commits just to trigger on-the-fly analysis of bots. We also suggest establishing two-way communication between bots and students/teaching staff to resolve some false positive warnings.

## BIBLIOGRAPHY

- [Akb18] Akbar, S. et al. "Poster: Improving Formation of Student Teams: A Clustering Approach". *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE. 2018, pp. 147–148.
- [Bal02] Baltes, B. B. et al. "Computer-mediated communication and group decision making: A meta-analysis". *Organizational behavior and human decision processes* **87.1** (2002), pp. 156–179.
- [Bec03] Beck, K. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [Bec15] Becker, K. et al. "Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment". *Information and Software Technology* **57** (2015), pp. 494–508.
- [Bel15] Beller, M. et al. "When, how, and why developers (do not) test in their IDEs". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 179–190.
- [BS16] Bhatia, S. & Singh, R. "Automated correction for syntax errors in programming assignments using recurrent neural networks". *arXiv preprint arXiv:1603.06129* (2016).
- [Bis16] Bishop, J. et al. "How to use open source software in education". *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM. 2016, pp. 321–322.
- [Bor17] Borle, N. C. et al. "Analyzing the effects of test driven development in GitHub". *Empirical Software Engineering* (2017), pp. 1–28.
- [Bru15] Bruegge, B. et al. "Software engineering project courses with industrial clients". *ACM Transactions on Computing Education (TOCE)* **15.4** (2015), p. 17.
- [Buc06] Buchta, J. et al. "Teaching evolution of open-source projects in software engineering courses". *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE. 2006, pp. 136–144.
- [BE12] Buffardi, K. & Edwards, S. H. "Exploring influences on student adherence to test-driven development". *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM. 2012, pp. 105–110.
- [Cam13] Campbell, J. L. et al. "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement". *Sociological Methods & Research* **42.3** (2013), pp. 294–320.
- [Can06] Canfora, G. et al. "Evaluating advantages of test driven development: a controlled experiment with professionals". *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM. 2006, pp. 364–371.

- [CK03] Carrington, D. & Kim, S.-K. "Teaching software design with open source software". *Frontiers in Education, 2003. FIE 2003 33rd Annual*. Vol. 3. IEEE. 2003, S1C–9.
- [Cau12a] Causevic, A. et al. "Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment." *XP*. Springer. 2012, pp. 138–152.
- [Cau12b] Causevic, A. et al. "Test case quality in test driven development: A study design and a pilot experiment" (2012).
- [Des09] Desai, C. et al. "Implications of integrating test-driven development into CS1/CS2 curricula". *ACM SIGCSE Bulletin*. Vol. 41. 1. ACM. 2009, pp. 148–152.
- [Dou05] Douce, C. et al. "Automatic test-based assessment of programming: A review". *Journal on Educational Resources in Computing (JERIC)* **5.3** (2005), p. 4.
- [Edw03a] Edwards, S. H. "Rethinking computer science education from a test-first perspective". *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM. 2003, pp. 148–155.
- [Edw03b] Edwards, S. H. "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance". *Proceedings of the international conference on education and information systems: technologies and applications EISTA*. Vol. 3. 2003.
- [ES14] Edwards, S. H. & Shams, Z. "Do student programmers all tend to write the same software tests?" *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM. 2014, pp. 171–176.
- [Ell07] Ellis, H. J. et al. "Can humanitarian open-source software development draw new students to CS?" *ACM SIGCSE Bulletin*. Vol. 39. 1. ACM. 2007, pp. 551–555.
- [Ell12] Ellis, H. J. et al. "Student software engineering learning via participation in humanitarian FOSS projects". *American Society for Engineering Education*. American Society for Engineering Education. 2012.
- [Ell15] Ellis, H. J. et al. "Team Project Experiences in Humanitarian Free and Open Source Software (HFOSS)". *ACM Transactions on Computing Education (TOCE)* **15.4** (2015), p. 18.
- [Ell08] Ellis, H. et al. "WIP: Challenges to educating students within the community of open source software for humanity". *The 2008 Frontiers in Education Conference*. 2008.
- [Erd05] Erdogmus, H. et al. "On the effectiveness of the test-first approach to programming". *IEEE Transactions on software Engineering* **31.3** (2005), pp. 226–237.

- [Fal14] Falkner, N. et al. "Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units". *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM. 2014, pp. 9–14.
- [FP12] Fox, A. & Patterson, D. "Crossing the software education chasm". *Communications of the ACM* **55.5** (2012), pp. 44–49.
- [FT13] Fucci, D. & Turhan, B. "A replicated experiment on the effectiveness of test-first development". *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE. 2013, pp. 103–112.
- [Fuc14] Fucci, D. et al. "Impact of process conformance on the effects of test-driven development". *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM. 2014, p. 10.
- [Fuc17] Fucci, D. et al. "A dissection of the test-driven development process: does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering* **43.7** (2017), pp. 597–614.
- [Geh07] Gehringer, E. et al. "Reusable learning objects through peer review: The Expertiza approach". *Innovate: Journal of Online Education* **3.5** (2007), p. 4.
- [Geh11] Gehringer, E. F. "From the manager's perspective: Classroom contributions to open-source projects". *Frontiers in Education Conference (FIE), 2011*. IEEE. 2011, F1E–1.
- [GW03] George, B. & Williams, L. "An initial investigation of test driven development in industry". *Proceedings of the 2003 ACM symposium on Applied computing*. ACM. 2003, pp. 1135–1139.
- [GW04] George, B. & Williams, L. "A structured experiment of test-driven development". *Information and software Technology* **46.5** (2004), pp. 337–342.
- [Ger12] Gerdes, A. et al. "An interactive functional programming tutor". *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM. 2012, pp. 250–255.
- [Ger17] Gerdes, A. et al. "Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback". *International Journal of Artificial Intelligence in Education* **27.1** (2017), pp. 65–100.
- [Gou15] Gousios, G. et al. "Work practices and challenges in pull-based development: the integrator's perspective". *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 358–368.
- [Gul14] Gulwani, S. et al. "Feedback generation for performance problems in introductory programming assignments". *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 41–51.



- [Hec18] Heckman, S. et al. “10+ years of teaching software engineering with itrust: the good, the bad, and the ugly”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM. 2018, pp. 1–4.
- [Hil16] Hilton, M. et al. “Tddviz: Using software changes to understand conformance to test driven development”. *International Conference on Agile Software Development*. Springer. 2016, pp. 53–65.
- [HG19] Hu, Z. & Gehringer, E. “Use Bots to Improve GitHub Pull-Request Feedback”. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM. 2019, pp. 1262–1263.
- [Hu18] Hu, Z. et al. “Open-source software in class: students’ common mistakes”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM. 2018, pp. 40–48.
- [HT00] Hunt, A. & Thomas, D. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.
- [Iha10] Ihantola, P. et al. “Review of recent systems for automatic assessment of programming assignments”. *Proceedings of the 10th Koli calling international conference on computing education research*. ACM. 2010, pp. 86–93.
- [JS08] Janzen, D. & Saiedian, H. “Test-driven learning in early programming courses”. *ACM SIGCSE Bulletin*. Vol. 40. 1. ACM. 2008, pp. 532–536.
- [JS06] Janzen, D. S. & Saiedian, H. “On the influence of test-driven development on software design”. *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*. IEEE. 2006, pp. 141–148.
- [Jon04] Jones, C. G. “Test-driven development goes to school”. *Journal of Computing Sciences in Colleges* **20.1** (2004), pp. 220–231.
- [Keu16] Keuning, H. et al. “Towards a systematic review of automated feedback generation for programming exercises”. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 2016, pp. 41–46.
- [KI08] Kollanus, S. & Isomöttönen, V. “Understanding tdd in academic environment: experiences from two experiments”. *Proceedings of the 8th International Conference on Computing Education Research*. ACM. 2008, pp. 25–31.
- [Kou07] Kou, H. “Automated inference of software development behaviors: Design, implementation and validation of zorro for test-driven development”. PhD thesis. University of Hawaii at Manoa, 2007.
- [Kou10] Kou, H. et al. “Operational definition and automated inference of test-driven development with Zorro”. *Automated Software Engineering* **17.1** (2010), p. 57.

- [Lat14] Latorre, R. "Effects of developer experience on learning and applying Unit Test-Driven Development". *IEEE Transactions on Software Engineering* **40.4** (2014), pp. 381–395.
- [Lau04] Laurent, A. M. S. *Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software*. " O'Reilly Media, Inc.", 2004.
- [Liu05] Liu, C. "Enriching software engineering courses with service-learning projects and the open-source approach". *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 613–614.
- [McC12] McCartney, R. et al. "Evaluating an early software engineering course with projects and tools from open source software". *Proceedings of the ninth annual international conference on International computing education research*. ACM. 2012, pp. 5–10.
- [McC57] McCracken, D. D. *Digital computer programming*. John Wiley & Sons, 1957.
- [Mis08] Mishali, O. et al. "The TDD-guide training and guidance tool for test-driven development". *International Conference on Agile Processes and Extreme Programming in Software Engineering*. Springer. 2008, pp. 63–72.
- [MH07] Müller, M. M. & Höfer, A. "The effect of experience on the test-driven development process". *Empirical Software Engineering* **12.6** (2007), pp. 593–615.
- [Nas15] Nascimento, D. M. et al. "Open source projects in software engineering education: a mapping study". *Computer Science Education* **25.1** (2015), pp. 67–114.
- [Obe09] Oberhauser, R. "Towards automated test practice detection and governance". *Advances in System Testing and Validation Lifecycle, 2009. VALID'09. First International Conference on*. IEEE. 2009, pp. 19–24.
- [PC11] Pančur, M. & Ciglarič, M. "Impact of test-driven development on productivity, code and tests: A controlled experiment". *Information and Software Technology* **53.6** (2011), pp. 557–573.
- [Pin17] Pinto, G. et al. "Training software engineers using open-source software: the professors' perspective". *The 30th IEEE Conference on Software Engineering Education and Training*. 2017, pp. 1–5.
- [RK06] Raj, R. K. & Kazemian, F. "Using open source software in computer science courses". *Frontiers in Education Conference, 36th Annual*. IEEE. 2006, pp. 21–26.
- [Sin13] Singh, R. et al. "Automated feedback generation for introductory programming assignments". *ACM SIGPLAN Notices* **48.6** (2013), pp. 15–26.
- [SA07] Siniaalto, M. & Abrahamsson, P. "A comparative case study on the impact of test-driven development on program design and test coverage". *Empirical Software Engineering*

- and Measurement, 2007. ESEM 2007. First International Symposium on.* IEEE. 2007, pp. 275–284.
- [Smi14] Smith, T. M. et al. “Selecting open source software projects to teach software engineering”. *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM. 2014, pp. 397–402.
  - [SLB08] Solar-Lezama, A. & Bodik, R. *Program synthesis by sketching*. Citeseer, 2008.
  - [Son15a] Song, Y. et al. “Closing the Circle: Use of Students’ Responses for Peer-Assessment Rubric Improvement”. *International Conference on Web-Based Learning*. Springer. 2015, pp. 27–36.
  - [Son15b] Song, Y. et al. “Pluggable reputation systems for peer review: A web-service approach”. *Frontiers in Education Conference (FIE), 2015 IEEE*. IEEE. 2015, pp. 1–5.
  - [Son16] Song, Y. et al. “Toward Better Training in Peer Assessment: Does Calibration Help?” *EDM (Workshops)*. 2016.
  - [Son17] Song, Y. et al. “Collusion in educational peer assessment: How much do we need to worry about it?” *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2017, pp. 1–8.
  - [SP06] Spacco, J. & Pugh, W. “Helping students appreciate test-driven development (TDD)”. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pp. 907–913.
  - [Sut14] Sutskever, I. et al. “Sequence to sequence learning with neural networks”. *Advances in neural information processing systems*. 2014, pp. 3104–3112.
  - [Tot06] Toth, K. “Experiences with open source software engineering tools”. *IEEE software* **23.6** (2006).
  - [Tsa14] Tsay, J. et al. “Influence of social and technical factors for evaluating contribution in GitHub”. *Proceedings of the 36th international conference on Software engineering*. ACM. 2014, pp. 356–366.
  - [Wil12] Wilkerson, J. W. et al. “Comparing the defect reduction benefits of code inspection and test-driven development”. *IEEE transactions on software engineering* **38.3** (2012), pp. 547–560.
  - [XL09] Xu, S. & Li, T. “Evaluation of test-driven development: An academic case study”. *Software Engineering Research, Management and Applications 2009* (2009), pp. 229–238.