A Test-Driven Approach to Improving Student Contributions to Open-Source Projects

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

2nd Yang Song
Department of Computer Science
University of North Carolina Wilmington
Wilmington, USA
songy@uncw.edu

3rd Edward F. Gehringer
Department of Computer Science
North Carolina State University
Raleigh, USA
efg@ncsu.edu

***Abstract*—Test-driven development (TDD) promises to help students write high-quality code with fewer defects. Although many studies of TDD usage have been conducted in entry-level computer science courses, few have looked at more advanced students doing projects of larger scope, such as contributing to open-source software (OSS). 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 a 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.**

***Index Terms*—Test-driven development, open-source software, software engineering, open-source curriculum**

I. 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* [1]. 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" [2]. 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 [3]–[7]. Several researchers conducted structured experiments and evaluated the effectiveness of the test-driven approach for entry-level computer science courses [8], [9]. Some experimental results showed that the test-driven approach can help developers write high-quality code and improve design skills [10], [11]. 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 [8], [12]. There is evidence that, compared with traditional development methods, the test-driven approach is no more effective in eliminating defects [13]. However, previous experiments were based on throwaway projects—projects used one time and discarded once finished [14].

This paper reports on a quasi-experimental controlled study based on an Open-Source Software (OSS) project named Expertiza, which is an open-source online peer-assessment tool [15]. Expertiza 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.

II. Related Work

A. 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

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

entry-level courses have fewer defects when the test-driven approach is used. Edwards [16] 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 [3].

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 [11] 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 and Saiedian [17] 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 and Isomöttönen [18] 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 [9] 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* [19]. They found that test cases created by the test-driven approach and the traditional test-last approach have almost the same quality.

B. Challenges of Adherence to the TDD Process

Previous studies showed that it was not easy to adhere to the TDD process all the time [20], [21]. Beller [21] 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 [22] 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 [23]. 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 [24] 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 paper, we change the focus of the TDD-like concept from the version control system to steps recorded by IDE plugins.

III. 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)?

A. Expertiza

The open-source contributions we examined were for an OSS project named Expertiza [15]. It is an open-source online peer-assessment tool initially funded by NSF. Since 2007, Expertiza has become the main source of course projects in for our masters-level course, Object-Oriented Design and Development. The application has been used by our university courses, and in courses at 21 other institutions around the world. The Ruby on Rails codebase is available on GitHub [25]. Expertiza has had more than 360 contributors, most of whom are students. Student contributions have helped Expertiza undergo several major updates and last for more than 10 years. Expertiza uses RSpec [26] as the automated testing framework, which is designed for TDD and is extensively used in industry.

B. OSS-based Course Projects

This study focuses on course projects in which students work on OSS contributions. The goals of these course projects are to understand the codebase, optimize current code and fix some glitches. These Expertiza projects are done in teams—of 2–3 members. Students are asked to choose which projects they are interested in, and are allowed (but not required) to form teams before indicating their preference. They register their preferences in Expertiza, and an intelligent algorithm [27] matches students and teams with projects. Students are allowed

to switch teams and projects if they don't like the team or topic they have been assigned.

C. 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.

Teaching and Tutorial Videos: Before starting the projects, all students studied the test-driven approach and RSpec testing framework for homework. In addition, the lead author recorded a series of tutorial videos [28] to help students understand the syntax of the RSpec testing framework and to demonstrate how to refactor the Expertiza codebase while adhering to the test-driven approach. The last video ended by following links to several exemplary RSpec test files for the students' reference.

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.

Test Skeletons: Test skeletons are test methods that have signatures (specifications) but not test bodies. According to previous research, tests generated by students could be shallow—tests concentrating on irrelevant or unlikely-to-fail conditions [14], also known as "happy path" tests [29]. 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 Fig. 1 (a). And we prepared two scenarios in test skeleton shown in Fig. 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 Fig. 1 (c) and make the tests pass. It is worth noting that tests can be much more complicated than the example in Fig. 1. Students are also responsible for writing tests for newly-created methods using the test-driven approach.

```
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)

Fig. 1: An example of the method that needs to be tested (a), corresponding test skeleton (b) and completed tests (c)

Before students started working on course projects, the lead author cleaned up some existing shallow tests and wrote tests for all the projects. Then he 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 used to check the correctness of students' modifications.

D. Experiment Design

We conducted our controlled study in the 2017 fall semester. We specified 12 course projects [30]. 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 [31], which is an automated code analysis tool. Students were asked to optimize codebase

TABLE I: Patterns followed in developing code and tests

Category	Pattern[a]
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 (*include*, *require*, etc.).
TDD[b]	Test modification$_1$ $\rightarrow$ Code refactoring$_1$ $\rightarrow$ ... [$\rightarrow$ Test configuration$_i$] $\rightarrow$ Test modification$_m$[$\rightarrow$ Code configuration$_j$] $\rightarrow$ Code refactoring$_n$[$\rightarrow$ Test modification$_{m+1}$]
Test-last[b]	Code refactoring$_1$ $\rightarrow$ Test modification$_1$ $\rightarrow$ ... [$\rightarrow$ Code configuration$_j$] $\rightarrow$ Code refactoring$_n$[$\rightarrow$ Test configuration$_i$] $\rightarrow$ Test modification$_m$[$\rightarrow$ Code refactoring$_{n+1}$]

[a] Subscript *m*, *n*, *i*, *j* represent the order of different categories. The category in square brackets indicates that it is optional.
[b] 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.

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 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.

E. Examination of Compliance with the TDD Process

In our experiments, we first used a RubyMine plugin named *Activity Tracker* [32] to record each student's development steps and store them in csv files. Each line in csv file records five data items: (1) timestamp, (2) current user, (3) keyboard events (character addition and deletion), (4) file opens in the text editor, (5) current location of the cursor (line number and column number). Based on source code and development steps, we wrote a script to generate intermediate files, which show the code after each development step by each student. Then we semi-automatically aggregated adjacent steps into *periods* and classified them into six categories, namely, test modification, test configuration, code refactoring, code configuration, TDD, and test-last. Table I shows the pattern used to identify each category. This approach is informed by the heuristics presented in Kou [22] and Fucci [33].

IV. 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 [34], 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 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 [14]) 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.

V. Data Analysis

A. 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). We deployed the post-survey to both TDD and non-TDD group. However, some survey questions were

TABLE II: Merge status and common mistakes of 12 course projects

Group	**TDD group**						**non-TDD group**					
Project id	1	2	3	4	5	6	7	8	9	10	11	12
Merge status	P	M	P	M	R	R	P	M	M	R	P	P
Common mistake ids[a]	3, 4	–	5	–	1, 3, 10, 11, 13	3, 5, 13	2	–	–	1, 2, 3, 8	3, 5	1, 5

[a] Thirteen common mistake ids with names: 1. Commenting, 2. Shallow/no tests, 3. Bad naming, 4. Duplicated code, 5. Failing functionality, 6. No pull request/not forked, 7. Hardcoding, 8. Messy pull requests, 9. Long methods, 10. Bad specifications, 11. Not following the existing design, 12. Limited modifications, 13. Switch statements.

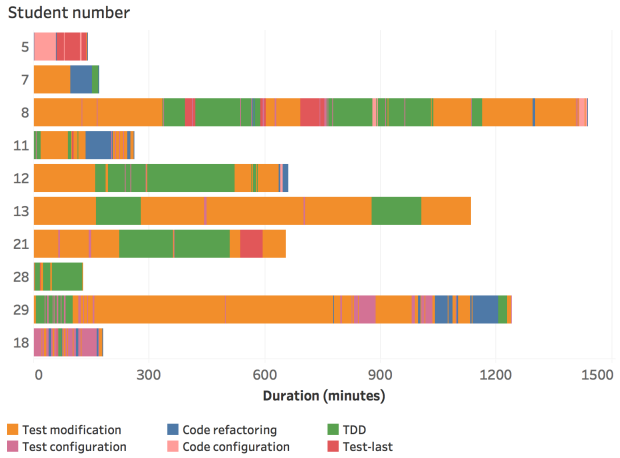

Fig. 2: An overview of student development steps

only applicable to the TDD group. More than half of students thought RSpec syntax 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."*

B. Compliance with the TDD Process

Fig. 2 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.

C. Quality Analysis of Source-Code Changes

Table II 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 [14]. 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.

```ruby
# user.rb
class User < ActiveRecord::Base
  def get_user_list
    ...
    if self.role.instructor?
      ...
      assgts=Assignment.where(instructor_id: self.id)
      ...
    end
    ...
  end
end
```

(a)

```ruby
# user.rb
class User < ActiveRecord::Base
  def get_user_list
    ...
    return Instructor.get_user_list if self.role.
        instructor?
    ...
  end
end

# instructor.rb
class Instructor < User
  def self.get_user_list
    ...
    assgts=Assignment.where(instructor_id: self.id)
    ...
  end
end
```

(b)

```ruby
# user_spec.rb
describe '#get_user_list' do
  ...
  allow(Instructor).to receive(:get_user_list).
      and_return([user1, user2])
  ...
end
```

(c)

```ruby
# user.rb
class User < ActiveRecord::Base
  def get_user_list; end
end

# instructor.rb
class Instructor < User
  def get_user_list
    ...
    assgts=Assignment.where(instructor_id: self.id)
    ...
  end
end
```

(d)

Fig. 3: 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)

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 Fig. 3 (a). Thus, the object that the *self* keyword refers to can be accessed by different *if* blocks. Fig. 3 (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 Fig. 3 (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 Fig. 3 (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 Fig. 3 (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* 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. Fig. 3 (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.

D. Test Quality Analysis

Table III 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 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) [35]. 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 shallow 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 source 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.

VI. 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 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

TABLE III: Overview of pre-existing (p)[a], staff-authored (t)[b] and student-authored tests (s)[c]

	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

[a] The data is based on pre-existing tests only.
[b] The data is based on pre-existing and staff-authored tests.
[c] The data is based on pre-existing and student-authored tests.

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 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.

VII. Threats to Validity

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 fixes some glitches, instead of implement 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.

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.

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.

VIII. 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. In the future, we will pay more attention to unit tests and integration tests. Also, we plan to upgrade the existing Internet bots [36] for GitHub pull requests to provide more instant and more accurate feedback to help students resolve common mistakes and code smells.

REFERENCES

[1] D. D. McCracken, *Digital computer programming*. John Wiley & Sons, 1957.

[2] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] S. H. Edwards, "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance," in *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, vol. 3, 2003.

[4] J. Spacco and W. Pugh, "Helping students appreciate test-driven development (tdd)," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 907–913.

[5] D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *ACM SIGCSE Bulletin*, vol. 40, no. 1. ACM, 2008, pp. 532–536.

[6] C. G. Jones, "Test-driven development goes to school," *Journal of Computing Sciences in Colleges*, vol. 20, no. 1, pp. 220–231, 2004.

[7] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on software Engineering*, vol. 31, no. 3, pp. 226–237, 2005.

[8] M. Pančur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Information and Software Technology*, vol. 53, no. 6, pp. 557–573, 2011.

[9] A. Causevic, D. Sundmark, and S. Punnekkat, "Test case quality in test driven development: A study design and a pilot experiment," 2012.

[10] B. George and L. Williams, "A structured experiment of test-driven development," *Information and software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[11] C. Desai, D. S. Janzen, and J. Clements, "Implications of integrating test-driven development into cs1/cs2 curricula," in *ACM SIGCSE Bulletin*, vol. 41, no. 1. ACM, 2009, pp. 148–152.

[12] M. Siniaalto and P. Abrahamsson, "A comparative case study on the impact of test-driven development on program design and test coverage," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 275–284.

[13] J. W. Wilkerson, J. F. Nunamaker, and R. Mercer, "Comparing the defect reduction benefits of code inspection and test-driven development," *IEEE transactions on software engineering*, vol. 38, no. 3, pp. 547–560, 2012.

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

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

[16] S. H. Edwards, "Rethinking computer science education from a test-first perspective," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003, pp. 148–155.

[17] D. S. Janzen and H. Saiedian, "On the influence of test-driven development on software design," in *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*. IEEE, 2006, pp. 141–148.

[18] S. Kollanus and V. Isomöttönen, "Understanding tdd in academic environment: experiences from two experiments," in *Proceedings of the 8th International Conference on Computing Education Research*. ACM, 2008, pp. 25–31.

[19] R. Martin, "The bowling game kata," *Site: http://butunclebob. com/ArticleS. UncleBob. TheBowlingGameKata*, 2005.

[20] M. M. Müller and A. Höfer, "The effect of experience on the test-driven development process," *Empirical Software Engineering*, vol. 12, no. 6, pp. 593–615, 2007.

[21] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 179–190.

[22] H. Kou, P. M. Johnson, and H. Erdogmus, "Operational definition and automated inference of test-driven development with zorro," *Automated Software Engineering*, vol. 17, no. 1, p. 57, 2010.

[23] H. Kou, "Automated inference of software development behaviors: Design, implementation and validation of zorro for test-driven development," Ph.D. dissertation, University of Hawaii at Manoa, 2007.

[24] N. C. Borle, M. Feghhi, E. Stroulia, R. Greiner, and A. Hindle, "Analyzing the effects of test driven development in github," *Empirical Software Engineering*, pp. 1–28, 2017.

[25] "Expertiza," 2018, https://github.com/expertiza/expertiza.

[26] "Rspec: Behaviour driven development for ruby," 2018, http://rspec.info.

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

[28] "Rspec tutorial videos," 2018, https://goo.gl/3dECtY.

[29] S. H. Edwards and Z. Shams, "Do student programmers all tend to write the same software tests?" in *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 2014, pp. 171–176.

[30] "Twelve course projects," 2019, https://github.com/Winbobob/A-Test-Driven-Approach-to-Improving-Student-Contributions-to-Open-Source-Projects/blob/master/README.md.

[31] "Codeclimate," 2018, https://codeclimate.com/github/expertiza/expertiza.

[32] "Activity tracker," 2018, https://plugins.jetbrains.com/plugin/8126-activity-tracker.

[33] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2017.

[34] "Pre-survey and post-survey," 2019, https://github.com/Winbobob/A-Test-Driven-Approach-to-Improving-Student-Contributions-to-Open-Source-Projects/tree/master/surveys.

[35] "Mutation score indicator," 2018, https://github.com/humbug/humbug#the-metrics.

[36] Z. Hu and E. Gehringer, "Use bots to improve github pull-request feedback," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 1262–1263.