Preprint: Paper to appear in ITiCSE 2021, June 26–July 1, 2021

Teaching Testing with Modern Technology Stacks in Undergraduate Software Engineering Courses

Scott P. Chow
scottpchow@ucsb.edu
Dept. of Computer Science
Santa Barbara, CA, USA

Tanay Komarlu
tkomarlu@ucsb.edu
Dept. of Computer Science
Santa Barbara, CA, USA

Phillip T. Conrad
phtcon@ucsb.edu
Dept. of Computer Science
Santa Barbara, CA, USA

ABSTRACT

Students' experience with software testing in undergraduate computing courses is often relatively shallow, as compared to the importance of the topic. This experience report describes introducing industrial-strength testing into CMPSC 156, an upper division course in software engineering at UC Santa Barbara . We describe our efforts to modify our software engineering course to introduce rigorous test-coverage requirements into full-stack web development projects, requirements similar to those the authors had experienced in a professional software development setting. We present student feedback on the course and coverage metrics for the projects. We reflect on what about these changes worked (or didn't), and provide suggestions for other instructors that would like to give their students a deeper experience with software testing in their software engineering courses.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → **Software development process management**.

KEYWORDS

testing; unit testing; integration testing; continuous integration; test coverage; web applications; computer science education; software engineering education

ACM Reference Format:
Scott P. Chow, Tanay Komarlu, and Phillip T. Conrad. 2021. Teaching Testing with Modern Technology Stacks in Undergraduate Software Engineering Courses. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26–July 1, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3430665.3456352

1 INTRODUCTION

This experience report describes introducing industrial-strength testing into CMPSC 156, an upper division course in software engineering at UC Santa Barbara (https://ucsb-cs156.github.io). A major learning goal of our course is to give students authentic experiences with industrial software development practices. This course had previously introduced such practices as coding in teams, working with legacy code, using version control, and various aspects of the Agile software design lifecycle methodology. However, one notably deficient practice was the depth of our students' understanding of software testing.

The authors (an instructor, and two TAs) had positive prior experiences with testing in industry settings where it was seen not as a burden, but as freeing, in the following sense: a solid test suite helps to reduce fear. With a solid test suite in place, developers are more confident about refactoring code to improve maintainability.

We became painfully aware of the gap between what we were *saying* to the students about the importance of testing, in contrast to what we were *implicitly teaching* about testing by its omission from our actual practice during the project phase of our course. We resolved that this needed to change. We wanted our students to come away, not just with an understanding of state of the art testing practices, but with a positive attitude towards testing and their own ability to write tests.

The literature describes many efforts made to improve teaching practices related to testing [3, 13, 15, 16, 28, 39, 41], but relatively few of these describe integrating testing into complex applications [21, 29] such as full-stack web applications. Early-career developers interviewed in [10] noted that the testing taught in courses differs significantly from that in industry; coursework often comes with limited scope and/or pre-written tests, while projects in industry are typically broader in scope and have complex dependencies.

We committed ourselves to the following goal: to transform the Fall 2020 offering of this course, to the extent possible, to one where all programming assignments and term projects would incorporate at least as much testing rigor as we had experienced in our internships in industry. This paper describes our experience with this transformation, reports on the feedback from our students, and makes recommendations for others that might seek to incorporate more testing into their own software engineering courses.

2 RELATED WORK

A number of articles were published in the early 2000s calling for testing to be more integrated into computer science curricula. In 2001, an article by Shepard et al. [36] criticized a lack of testing in undergraduate curricula, citing the famous claim from Fred Brooks' "The Mythical Man Month" [26]—that 50% of project development time will be spent on testing. While more recent research in developer behaviors from Beller [6] puts the fraction of time spent on testing at 25%, in any case testing makes up a significant portion of the development lifecycle. Articles by Jones [24, 25] and Christensen [8] suggested that, given the already high saturation of topics in computer science curricula, testing should instead be

ITiCSE 2021, June 26–July 1, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8214-4/21/06.
https://doi.org/10.1145/3430665.3456352

rolled into every single topic being taught. This would later be supported by work from Desai et al. [13] in 2008, which demonstrated it was possible to integrate testing into introductory courses without severely inflating student coursework.

Work from Sun and Jones [39] demonstrated that it was feasible to teach students to build and test GUI applications, while work from Thornton et al. [41] demonstrated that student testing of GUI programs may have benefited from the visual nature of the application. Our work serves a similar purpose, but with full stack web applications instead of GUI applications.

It is worth noting that the flavor of testing being advocated for was "test-first" or test driven development (TDD), which is defined by Kent Beck [4, 5]. The claimed benefits of TDD include (1) increased confidence in code behavior, (2) easier and more confident refactoring, and (3) more thorough understanding of design choices.

Tinkham and Kaner [42] found evidence that their students struggled to adhere to TDD when the tasks defined were too simple, but that student work improved as a result of it. On the other hand, Kollanus and Isomöttönen [29] found that making the task too complex hampered students' ability to adhere to TDD, as they became confused on how to test the different components. Kollanus and Isomöttönen also found that the testing tools' ease of use played a key role in helping students adhere to TDD. In addition, a more recent study from Kazerouni et al. [28] suggests that, while incremental testing is positively correlated with code coverage, "test-first" TDD is negatively correlated with coverage. This work suggests that not only is there a balancing act with regards to complexity and adherence to TDD, but that "test-first" TDD might also be incompatible with the metric of code quality we pursue in this work.

The conclusion about the relationship between testing tools' ease of use and adoption are further supported by more recent work from Wrenn and Krishnamurthi [43], which explores encouraging student adoption of testing practices via prompting students to produce high quality input-output examples prior to implementing an assignment through their IDE. While not definitive, Wrenn and Krishnamurthi produce promising results demonstrating that students were more willing to engage in example creation prior to implementation without instructor coercion when they are prompted to do so by their IDE.

Furthermore, a paper by Avellar et al. [3] provides evidence that correctness of student submissions increases significantly when students are encouraged to test as opposed to not. While the value of teaching testing to students has been reproduced multiple times over the past two decades, getting students to adhere to testing can represent a significant challenge.

As an aside, gamification was also explored as an avenue to help students test and comply with TDD. While findings from Fraser [18] primarily focus on gamifying testing as a pedagogical tool, they do briefly explore the potential of gamifying software testing practices via achievement systems.

Hand in hand with testing comes the concept of automated feedback and grading, such as Edwards' work on Web-CAT, an online submission system that provides automated feedback and grading on student programming assignments [15]. The initial student feedback on Web-CAT was largely positive, with students generally recognizing it as a useful tool in their development workflow. Later work from Buffardi, Edwards, and Pérez-Quiñones [7, 16] further supports this claim. Edwards and Shams [17] introduced an "all-pairs" method of evaluating the quality of tests, where a student's test suite is run against all other student submissions in an attempt to screen other students' submissions for bugs. Work from Hu et al. [21] also suggests that students are amenable to receiving automated feedback from bots. A majority of students in that paper felt that the bot's feedback helped improve their contributions.

Our work is most similar to that of Krutz et al. [30] which describes an undergraduate course focused on software testing; we share the motivation of providing students with an authentic and deep encounter with software testing. The key difference between their work and ours is that in our course, testing is one topic among many, while their work describes an entire course centered around testing.

Another paper with similar themes to our work is that of Hu and Gehringer [21]; like us, their work describes the use of a continuous integration (CI) pipeline for automating feedback to students on aspects of their code. In their work, the CI pipeline provided feedback from both linting and style checking through static analysis, as well as results from running a test suite. Two differences between their work and ours is that they studied a graduate rather than undergraduate course, and they did not report on code coverage; only on passed vs. failed tests.

3 PRE-MODIFICATION COURSE OVERVIEW

This course is split into two phases: (1) a class with traditional assignments and labs to prepare students with basic Java and software engineering skills, and (2) developing and extending one of several legacy code projects.

Topics covered in the first half of the course include:

- Fundamental concepts of Java
- Basic testing and test coverage tools for Java
- Running and deploying a Spring Boot (Java) application
- Basics of web application development (communicating with a database, frontend-backend communication, etc.)

During the first half of the course, most programming assignments were autograded using JUnit [27] test suites written by course staff, while others were graded by interactive manual testing and code inspection. During the second half of the course, students were assigned to work on legacy code projects in teams of 4–6 students, each working on an epic—a term used in Agile for a group of related user stories organized around a theme. In this phase, when teams completed work on a user story, they would make a pull request (PR) from a feature branch into the default branch (i.e. `master` or `main`); PRs would trigger a round of code review and testing by course staff. Points towards their project grade were earned only when code review suggestions were made, and a PR was accepted and merged into the default branch.

Prior to the curriculum modifications described in this report, it was common that students would be unable to write any automated tests to verify the functionality of their features. Even if they were inclined to do so, often the course staff (the instructor and TAs) would struggle to provide guidance due to a lack of knowledge. The codebase contained few examples of tests that could be used as models. As a result, it was common to give credit for testing by

requiring students to make only token efforts at testing (i.e writing a single unit test for one component of their code). In practice, the only significant testing that was done was manual testing via live demos.

In a later section, we describe how we modified this course to utilize more autograding, teach students to write more meaningful tests, as well as how to test complex applications with multiple interdependent components, such as a full-stack web application.

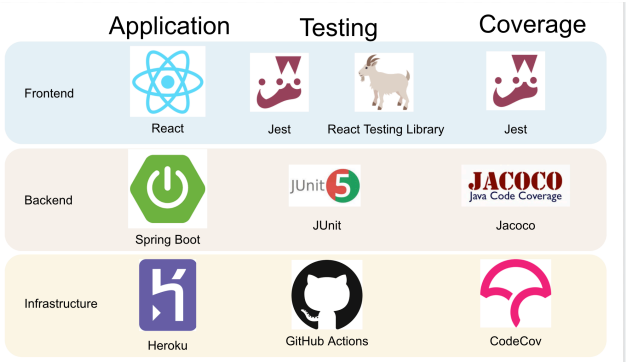

Figure 1: Tech Stack for Modified Course

4 TECH STACK OVERVIEW

We now provide an overview of the technology stack (see Figure 1) we use in this course in order to demonstrate how these technologies facilitate our goal of introducing students to testing technologies and practices.

4.1 Spring Boot and Maven

Spring Boot [37] is a Java framework for building complex applications that is capable of orchestrating a web server, application server, and database with little or no boilerplate or configuration code required. In this course, Spring Boot serves as our application's backend, publishing a RESTful API for our frontend to communicate with. This framework facilitates testing by eliminating the need for large swathes of boilerplate and connection code. The elimination of that code helps reduce both the amount and complexity of testing required, allowing students to focus on testing the behavior of the application. We use Maven [2] as our build tool.

4.2 React

React [33] is a "JavaScript library for building user interfaces." In this course, React is responsible for building a single page application (SPA) that communicates to the RESTful API published by our Spring Boot backend. React facilitates testing by providing well documented and maintained testing libraries such as React Testing Library [34] that simplify frontend testing. React Testing Library does this by abstracting away many of the complexities involved in testing frontend applications, such as the Document Object Model (DOM) as well as DOM Events. Furthermore, we have found React and its libraries to be well documented, with significant community presence on StackOverflow. This documentation and community presence help both students and instructional staff get up to speed on React as well as debug any issues that arise.

4.3 GitHub Actions

GitHub [19] is an online development platform that can, among many things, host code and facilitate project management. We utilize GitHub as an online repository and project management tool for student assignments and legacy code projects. In the modified version of the course, we take advantage of GitHub's Actions feature to provide continuous improvement (CI) workflows which automatically run tests on feature branches and pull requests, and calculated test coverage. This facilitates adherence to testing by serving as a continuous check that tests are passing and coverage is being maintained.

4.4 Testing and Coverage

For backend Java code we use JUnit [27] as our testing framework for Java code while using Jacoco [22] to generate code coverage reports. For the frontend JavaScript code, we use Jest [23], an industry standard testing framework that comes pre-configured with most React apps. Jest provides features for test automation and computing test coverage. Finally, we use CodeCov [9] to aggregrate results of the separate JUnit and Jest reports, and publish code coverage reports as comments and checks on pull requests.

It is worth noting that while work performed by Shams and Edwards [35] suggests alternatives to code coverage as a testing quality metric (such as checked coverage) we use code coverage as our primary metric for both testing quality and adherence to testing as a practice. This is primarily due to a lack of mature resources—such as Jacoco, and Jest—for implementing alternative testing coverage metrics in both Java and JavaScript.

Another available technique that measures test quality is mutation testing [12], which measures a test suite's ability to identify and kill mutants, or versions of the code with mutations (e.g. replacing a > with <=.) In the modified version of the course, we incorporated mutation testing using Pitest [32] into two of the traditional Java assignments. For the legacy code projects, we have so far incorporated only code coverage; incorporating mutation testing is left as future work.

There are limitations to relying on code coverage as the sole metric for measuring the quality of a test suite. In the extreme, it is possible to have 100% test coverage with no meaningful testing by not actually asserting any behaviors about the exercised code. Dijkstra said "program testing can be used very effectively to show the presence of bugs but never to show their absence" [14]. In the same spirit, a lack of coverage shows the absence of testing, but the presence of coverage does not guarantee the presence of testing. Furthermore, it can sometimes be detrimental to pursue 100% code coverage; it is not always possible to achieve and, in the extreme, can even be detrimental to the quality of the implementation code.

Thus, we do not rely on code-coverage alone, but also depend on instructional staff to code review every pull request; we verify that the test coverage is meaningful and that any gaps in coverage truly represent code that is unfeasible to test, rather than simply a lack of effort.

4.5 Heroku

Heroku [20] serves as the deployment and hosting platform for our web applications. Utilizing the no-cost tier of Heroku, we are able to create many independent deployments of our web applications by linking them to repositories on GitHub. We can then use these multiple deployments to create different levels of quality assurance between production and qa deployments. This facilitates manual testing of a live deployment by allowing students and staff to test new changes in a deployed environment that is not production.

4.6 Codebase Organization

We organized the code repositories with testing in mind. The project root follows the standard Maven directory structure (i.e. a top level `src` directory, with subdirectories `src/main/java` and `src/test/java` for application and test code respectively. Code for the JavaScript frontend is stored under a top level *javascript* directory; the files under *javascript* follow the directory conventions of a React application. While not required by the frameworks we used, we chose to also make a `javascript/src` and `javascript/test` division here for consistency.

Under both `src/main/java` and `javascript/src`, we further divide the code into different subdirectories depending on the code's general purpose. For example, for code that uses the Model-View-Controller design pattern, we have separate directories for Java classes representing models, views, and controllers. Similarly the Javascript code is divided among code for top level pages, reusable React components, and utility libraries of non-React JavaScript code.

We follow the usual Maven convention that each file under the `src/main/java` directory is in a 1-to-1 correspondence with a corresponding test file under `src/test/java`. While not a requirement of Jest, we imitate this convention under the `javascript/src` and `javascript/test` directories. The result is that there is a clear expectation that *every source code file in the project has a corresponding file of tests*. Further, since the code is divided functionally, when looking for an example of how to write or test a certain type of code (e.g. the controller code for the backend of a RESTful API, or the code for validation of a React form), the developer can typically find examples of similar code in the `src` or `test` directory to which they are contributing.

When writing tests, we use mocking and stubbing to test Java classes and React components independently of one another. For Java, we use the Mockito [31] framework for mocks, while for the JavaScript code, mocking is built into the Jest framework.

5 POST-MODIFICATION COURSE OVERVIEW

The first of our project goals was to modify the curriculum to include industry practices for testing. To achieve this goal, we:

- Created instructional materials to familiarize students with testing in both Java and JavaScript.
- Created an example application using the technology stack as a proof of concept.
- Re-implemented a minimal viable product of all legacy code projects using the new technology stack to incorporate testing and test coverage metrics. This includes updating pull request workflows to enforce testing and test coverage.

It should be noted that the course's shift to the new technology stack was happening independently of this project; we simply enhanced this change by adding testing and test coverage tools as a part of this shift.

5.1 Instructional Materials

To introduce students to testing and test coverage, we created a series of labs that began with students extending and testing a rational number calculator built in React. The goals of this lab were (1) introducing students to JavaScript and React and (2) teaching students how to run and test React applications. Students were provided with some unit tests examples, but were required to write additional tests in order to achieve the required level of coverage. It was straightforward to create an autograder for this assignment.

Another assignment involved developing "Plain Old Java Objects (POJOs) to represent items on a restaurant menu, and the menu itself. The goal of this lab was to have students learn about testing, code coverage, and mutation testing in Java. This lab was already part of the course, but the test coverage and mutation testing aspects were new with this course iteration; we based 70% of the students grade on instructor tests, and 30% on the fraction of mutants killed.

Finally, we assigned two labs to introduce students to running and deploying web applications. The first web application was a simple "Hello World" Spring Boot web application and the second one was a Todo application developed by the teaching staff. The goals of these labs were to introduce students to the steps required to run an application locally on their machines and the steps required to deploy those applications to Heroku. These steps included configuring their applications to authenticate using Google login credentials. We were able to introduce testing and code coverage—and therefore autograding—to these applications using JUnit and Jacoco, leveraging the same techniques from the previous labs.

To demonstrate the technology stack, we created a todo list manager web app; this app allowed each user to maintain a single todo list, unique to that user. This app demonstrated basic features common to many webapps, including simple Create/Read/Update/Destroy operations, authentication using OAuth, application navigation, and techniques for writing tests for each of these functions. This app served several purposes: (1) It provided the course staff an opportunity to acquaint themselves with the new technology stack. (2) It provided an opportunity to make certain decisions about how we would organize the codebase so as to maintain consistency across assignments and projects. (3) It became the basis for future labs and projects.

Basing the class materials and projects on a common codebase allowed us to develop them significantly faster than starting each project from scratch, and promote consistency among the various legacy code projects.

This application's code is open source and can be found on GitHub [1].

5.2 Reimplementing Legacy Code Projects

In order to transition our previous legacy code projects to the new technologies we were using, we decided to restart the development of these applications in the new technology stack.

We did this because we could not see a clear way to adapt the previous application stack to the new technology stack over the course of the quarter. In addition, the previous application had few if any tests. This meant that, if we did not restart the applications from scratch, much of the work students would have to do this quarter would only be writing tests for the existing code. While we want to provide an authentic experience that mirrors industry, we did not want to risk student engagement by forcing them to only write tests to pay down the accumulated technical debt of past quarters. We could also afford to develop these applications anew because we had the benefit of developing from a starter application. This allowed the course staff to present students with three different functioning applications with 100% test coverage to work on.

Because we adopted the new technology stack in each of our legacy applications, we were able to instrument them to track and enforce testing and code coverage; we used GitHub Actions to run the entire test suite on every push to the repository, and every pull request. The CodeCov's bot flags any changes that reduce code coverage as a "failing test". This mirrors industry practices and raises student's awareness of testing while writing code.

This enforcement of test coverage throughout all programming assignments in the course is the key difference that we examine in this experience report.

6 EVALUATION AND REFLECTIONS

In this section we describe the kinds of data we gathered for evaluation purposes, and then what worked well, and what didn't, citing our data where appropriate. We conclude the paper with advice for practitioners that may want to adopt all or part of our work.

6.1 Evaluation Data

We instrumented and tracked code coverage and test count across the three legacy code projects both before (Winter 2020) and after the course modifications (Fall 2020). All projects began Fall 2020 with 100% code coverage; we required students to maintain 100% test coverage in order to contribute to the projects throughout the quarter, except in cases where a good argument could be made for an exception. We provide these results in Table 1. In total, 707 new tests were added by 66 students, making for an average of 10.9 tests per student. This is compared to 0.2 tests per student from the Winter 2020 offering of this course, where writing automated tests was encouraged but not required. Due to a significant amount of group programming and students not listing group members as co-authors in their commits, we were unable to accurately attribute tests to students and determine the distribution of these tests across individual students in the class.

Project	Test Count		Test Coverage	
	Pre	Post	Pre	Post
Courses Search	71	394	30%	99.53%
CS Tutor Program	2	368	2%	100.00%
Mapache Search	6	341	1%	100.00%

Table 1: Test Statistics from Winter 2020 (Pre) and Fall 2020 (Post)

In addition, we conducted an anonymous survey of students (58 out of 66 students responded, an 87.9% response rate) that had just completed the Fall 2020 offering of the modified course, where we asked "What impact has this course had on how you view testing?".

We also conducted a focus group of five students from this class, asking them (1) what impact the course had on their attitudes towards testing, and (2) what suggestions they had for changes to the course in the future, specifically with respect to testing.

6.2 What Worked

The modified course succeeded in achieving our primary goal, which was to introduce testing with the same rigor that the authors had experienced in industry settings. This can be observed in how all but one legacy code project was able to maintain 100% code coverage with student contributions. To be clear, 100% code coverage does not mean that *all* code was covered, but that all code that was not excluded from coverage was covered. One example of code that was excluded was code that leveraged a third party package to manage authentication and redirection on the frontend. We excluded this code from coverage because the library's functionality failed when executed in a testing environment, and we were unable to find a work around.

The modified course represents a significant improvement over the previous iteration of the course. We acknowledge that this is not an apples-to-apples comparison, since we did not make testing a strict requirement in the previous iteration of the course. Nevertheless, we demonstrated that we were able to achieve our goal of introducing automated testing as a significant and meaningful aspect of the software development lifecycle.

In achieving that goal, we were also able to clarify how testing is achieved in a complex web applications, as well as provide an authentic experience with testing. In the post-course survey, one student described how *"Before this class, I had a vague idea about testing C++ programs, but was unsure about how that would translate to real-world web applications... However, I feel more comfortable writing my own test cases for Java functions and React components now, after practicing a lot in the group project and programming assignments. It certainly has made me view testing as a larger aspect of the software development cycle than I had realized before."* Another student explained that *"This course has shown me that testing is a necessary feature when working with larger programs and code bases. I have only worked with smaller programs in the past where a test suite seems needless."*

Furthermore our efforts with code organization also yielded benefits, with students able to identify testing examples and test locations based on a file's location. In the focus group, one student describes how *"the actual directory structure matched the files we're looking for so that was definitely very helpful"* because *"it was ... helpful that ... if I borrowed a table from some component that I knew exactly where to go to look for the corresponding tests for that table."* Another student in the focus group followed up with how *"the way that information was like structured like just the code that we were given was super helpful for both implementation and testing because sometimes... if I didn't understand how like a component was being used or like I didn't understand like what data do I need to pass here in this prop, I could go take a look at how that's been done in the*

past." These comments from the focus group also align with our experiences with teaching students to write tests for their projects: our first step would often be to pull up an example test from the layer under test or another related layer.

6.3 Areas for Improvement (What didn't work)

The biggest area for potential improvement is more effectively teaching testing before students encounter it in their projects. We required students to complete programming assignments involving testing before entering the project phase, however the tests we asked them to write were for relatively straightforward methods of "plain old Java Objects (POJOs)", and very simple React components. Students had difficulty applying what they had learned to a more authentic application development context, e.g. being assigned to add a particular feature to an application, where the feature was described only by a user story.

When asked how they might change the curriculum, three students from the focus group said they would include more programming assignments to teach testing in the context of the projects. A student from the focus group confessed that *"if I was trying to test [the project] on my own and if I didn't have the examples I wouldn't have been able to come up with [the course staff's] methods of doing what they did."* While we made an effort to explain in lecture the testing methods students would encounter in the projects, this feedback demonstrates that delivery via lecture alone is insufficient for student learning. Furthermore, a student from the focus group pointed out that they felt the programming assignments were too "cookbook," suggesting instead to modify the assignments to allow for more "learning by doing."

This student's feedback aligns with our anecdotal experience when helping students with testing their projects, as referencing steps from a programming assignment was often insufficient to jog the memory of a student of a particular concept.

In future iterations, we intend to provide examples of the entire process of developing tests starting from a user story, and in the context of one of our legacy code projects. This should be presented in multiple forms: in code, in a written narrative, and in video. Our testing assignments also need to better teach and explain the mechanics of the testing libraries and testing techniques that they will observe and use in their projects; while these assignments may start with simple POJOs and React Components, they need to progress to the complexity students will encounter in the course projects so that students gain both technical skills and self-confidence in their ability to write tests.

Anecdotally, the authors noticed that students seemed to struggle more with frontend testing than backend testing. The focus group feedback aligned with this observation; while describing testing of the behavior of a button in the frontend, one student explained *"it was a little bit annoying to have to go in and add tests for things that seemed inconsequential or kind of just obvious that it works"*. We conclude that we could do a better job teaching that the goal of testing is not only verifying that a behavior functions at the time of creation but for ensuring that behavior continues to function in a reproducible and automated manner.

6.4 Future Work and Recommendations

In this section we outline our intentions for future work and recommendations to practitioners that may wish to adopt part of all of the practices described in this paper.

Besides addressing the previously outlined areas for improvement, there are two directions for future work: (1) adding mutation testing and (2) end to end testing to the projects. Adding mutation testing to the projects, using tools such as Stryker Mutator [38] for JavaScript and Pitest [32] for Java, would complement code coverage in providing greater insight into the quality of our projects' test suites, by ensuring that the coverage does enforce some behaviors about the projects' functionality. Adding end to end testing to the projects, using tools such as Cypress [11] or TestCafe [40], would provide greater confidence in the functionality of the projects as a whole. Given that much of the testing done in modified version of the course was white box testing, this also provides an opportunity for students to engage in black box testing and allow students to experience the differences between the two.

For instructors that wish to adopt these practices, the most important thing they can do is to create an example project of their own, using the technologies and practices they want to adopt. This example project serves two purposes: (1) it provides the instructors and their teaching staff an opportunity to learn an experience the technology and practices themselves and (2) the example project can be used to create a baseline project from which instructors can create assignments and students can use as a basis for their own projects.

We also suggest instructors and their teams to pursue 100% code coverage when creating their example projects. While we have previously discussed the issues with pursuing 100% code coverage, the goal in this case is not necessarily to achieve it. Instead, the goal is to learn where achieving coverage becomes difficult and to develop patterns and policies for dealing with those difficulties. In the case where achieving complete coverage can be shown to not be the best course of action, it provides an example of the weaknesses of code coverage that students can learn from. In our projects, for example, we excluded library boilerplate code dealing with frontend authentication from coverage due to behaviors such as third-party OAuth authentication being difficult to mock in tests.

Finally, we would recommend performing this transition over multiple course iterations, beginning with the example application, then adding testing and code coverage to the assignments, and finally enforcing testing and code coverage on projects. While we are glad to have completed the transition quickly, it made for a stressful experience for both the teaching staff and the students. Performing the transition over multiple iterations may have yielded better results, with a more robust curriculum and thorough testing practices. It is certainly what we strive for in future iterations of this course.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation, awards 1915196 and 1915198. We also thank the students of UCSB's CMPSC 156, Fall 2020, the course staff: Mara Downing, Andrew Lu, Bryan Terce, Gabriel Soule, and Darragh Burke.

REFERENCES

[1] 2020. GitHub Repo: demo-spring-react-todo-app. https://github.com/ucsb-cs156-f20/demo-spring-react-todo-app

[2] Apache Maven Project 2020. Apache Maven Project. https://maven.apache.org/

[3] Gustavo M. N. Avellar, Rogério F. da Silva, Lilian P. Scatalon, Stevão A. Andrade, Márcio E. Delamaro, and Ellen F. Barbosa. 2019. Integration of Software Testing to Programming Assignments: An Experimental Study. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9. https://doi.org/10.1109/FIE43999.2019.9028519 ISSN: 2377-634X.

[4] Beck. 2002. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] K. Beck. 2001. Aim, fire [test-first coding]. *IEEE Software* 18, 5 (Sept. 2001), 87–89. https://doi.org/10.1109/52.951502 Conference Name: IEEE Software.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[7] Kevin Buffardi and Stephen H. Edwards. 2014. A formative study of influences on student testing behaviors. In *Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 597–602. https://doi.org/10.1145/2538862.2538982

[8] Henrik Bærbak Christensen. 2003. Systematic testing should not be a topic in the computer science curriculum!. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education (ITiCSE '03)*. Association for Computing Machinery, New York, NY, USA, 7–10. https://doi.org/10.1145/961511.961517

[9] Codecov 2020. Code Coverage Done Right. https://codecov.io

[10] Michelle Craig, Phill Conrad, Dylan Lynch, Natasha Lee, and Laura Anthony. 2018. Listening to Early Career Software Developers. *J. Comput. Sci. Coll.* 33, 4 (April 2018), 138–149.

[11] Cypress 2020. Cypress: JavaScript End to End Testing Framework. https://www.cypress.io

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136 Conference Name: Computer.

[13] Chetan Desai, David S. Janzen, and John Clements. 2009. Implications of integrating test-driven development into CS1/CS2 curricula. *ACM SIGCSE Bulletin* 41, 1 (March 2009), 148–152. https://doi.org/10.1145/1539024.1508921

[14] Edsger W. Dijkstra. n.d.. E.W. Dijkstra Archive: On the reliability of programs. (EWD303). (n.d.). http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF circulated privately.

[15] Stephen H. Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing* 3, 3 (Sept. 2003), 1–es. https://doi.org/10.1145/1029994.1029995

[16] Stephen H. Edwards and Manuel A. Pérez-Quiñones. 2007. Experiences using test-driven development with an automated grader. *Journal of Computing Sciences in Colleges* 22, 3 (Jan. 2007), 44–50.

[17] Stephen H. Edwards and Zalia Shams. 2014. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 354–363. https://doi.org/10.1145/2591062.2591164

[18] Gordon Fraser. 2017. Gamification of Software Testing. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. 2–7. https://doi.org/10.1109/AST.2017.20

[19] GitHub 2020. GitHub: Where the world builds software. https://github.com/

[20] Heroku 2020. Cloud Application Platform | Heroku. https://www.heroku.com/

[21] Zhewei Hu and Edward F. Gehringer. 2019. Improving Feedback on GitHub Pull Requests: A Bots Approach. In *2019 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE43999.2019.9028685 ISSN: 2377-634X.

[22] JaCoCo 2020. JaCoCo Java Code Coverage Library. https://jacoco.org/jacoco

[23] Jest 2020. Jest: Delightful JavaScript Testing. https://jestjs.io/

[24] Edward L. Jones. 2000. Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian conference on Computing education (ACSE '00)*. Association for Computing Machinery, New York, NY, USA, 153–157. https://doi.org/10.1145/359369.359392

[25] Edward L. Jones. 2001. Integrating testing into the curriculum — arsenic in small doses. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education (SIGCSE '01)*. Association for Computing Machinery, New York, NY, USA, 337–341. https://doi.org/10.1145/364447.364617

[26] Fredrick P. Brooks Jr. 1982. *The Mythical man-month : essays on software engineering*. Reading, Mass. : Addison-Wesley Pub. Co., 1982. ©1975. https://search.library.wisc.edu/catalog/999550146602121

[27] JUnit5 2020. JUnit 5. https://junit.org/junit5/

[28] Ayaan M. Kazerouni, Riffat Sabbir Mansur, Stephen H. Edwards, and Clifford A. Shaffer. 2019. Student Debugging Practices and Their Relationships with Project Outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1263. https://doi.org/10.1145/3287324.3293794

[29] Sami Kollanus and Ville Isomöttönen. 2008. Understanding TDD in academic environment: experiences from two experiments. In *Proceedings of the 8th International Conference on Computing Education Research (Koli '08)*. Association for Computing Machinery, New York, NY, USA, 25–31. https://doi.org/10.1145/1595356.1595362

[30] Daniel E. Krutz, Samuel A. Malachowsky, and Thomas Reichlmayr. 2014. Using a real world project in a software testing course. In *Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/2538862.2538955

[31] Mockito 2020. Mockito: Tasty mocking framework for unit tests in Java. https://site.mockito.org/

[32] Pitest 2020. PIT Mutation Testing. https://pitest.org/

[33] React 2020. React – A JavaScript library for building user interfaces. https://reactjs.org/

[34] React Testing Library [n.d.]. React Testing Library | Testing Library. https://testing-library.com/docs/react-testing-library/intro

[35] Zalia Shams and Stephen H. Edwards. 2015. Checked Coverage and Object Branch Coverage: New Alternatives for Assessing Student-Written Tests. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 534–539. https://doi.org/10.1145/2676723.2677300

[36] Terry Shepard, Margaret Lamb, and Diane Kelly. 2001. More testing should be taught. *Commun. ACM* 44, 6 (June 2001), 103–108. https://doi.org/10.1145/376134.376180

[37] Spring Boot 2020. Spring Boot. https://spring.io/projects/spring-boot

[38] Stryker 2020. Stryker Mutator. https://stryker-mutator.io/

[39] Yanhong Sun and Edward L. Jones. 2004. Specification-driven automated testing of GUI-based Java programs. In *Proceedings of the 42nd annual Southeast regional conference (ACM-SE 42)*. Association for Computing Machinery, New York, NY, USA, 140–145. https://doi.org/10.1145/986537.986570

[40] TestCafe 2020. TestCafe: A node.js tool to automate end-to-end web testing. https://devexpress.github.io/testcafe/

[41] Matthew Thornton, Stephen H. Edwards, Roy P. Tan, and Manuel A. Pérez-Quiñones. 2008. Supporting student-written tests of gui programs. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 537–541. https://doi.org/10.1145/1352135.1352316

[42] A. Tinkham and C. Kaner. 2005. Experiences teaching a course in programmer testing. In *Agile Development Conference (ADC'05)*. 298–305. https://doi.org/10.1109/ADC.2005.25

[43] John Wrenn and Shriram Krishnamurthi. 2020. Will Students Write Tests Early Without Coercion?. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3428029.3428060