Student difficulties in Unit Testing, Integration Testing and Continuous Integration: An exploratory pilot qualitative study

Bhuvana Gopal
Computer Science and Engineering
University of Nebraska-Lincoln
bhuvana.gopal@unl.edu

Stephen Cooper
Computer Science and Engineering
University of Nebraska-Lincoln
stephen.cooper@unl.edu

Justin Olmanson
College of Education and Human Sciences
University of Nebraska-Lincoln
jolmanson2@unl.edu

Ryan Bockmon
Computer Science and Engineering
University of Nebraska-Lincoln
ryan.bockmon@huskers.unl.edu

Abstract

In this exploratory pilot study, we trace students' experiences in designing, developing and deploying unit and integration tests, as well as setting up and running continuous integration, during a semester-long, industry-partnered software engineering course project. We conducted a qualitative study using semi-structured interviews, and combined them with researcher memos, and reflective researcher journals. In so doing, we identified various difficulties that our novice testers were plagued by during the testing and DevOps process. Some of those difficulties include: communication within the team and other stakeholders, prioritization of features to be tested, entry and exit criteria for tests, difficulties with learning tools associated with testing, the time commitment involved in designing, writing and implementing meaningful tests, not knowing what kind of questions to ask and of whom, and how to look for test completeness beyond code coverage. We investigate and discuss these difficulties in detail. We also explore the cognitive biases identified as impediments to these testers in accomplishing their testing goals.

1. Introduction

Software testing is an important aspect of creating reliable software. Recent studies have indicated that testing makes up over 50% of the cost of a typical software project (Osterweil, 1996) and that a third of the cost of all software bugs could be eliminated through improved testing (Wong et al., 2011).

1.1. Conceptual Orientation and Theoretical Framework

The curricular orientation of many schools is to teach students how to build software, not break it (Meneely & Lucidi, 2013). This leads to testing often being only a minor portion of a student's grade in the majority of programming courses (Jones & Chatmon, 2001; Smith, Tessler, Kramer, & Lin, 2012). Students are rarely enthusiastic about testing; they are usually much more interested in building software than in ensuring its quality (Clark, 2004). There is a significant body of research showing that software testing in our educational institutions requires a heightened and more practical focus–with educators doing a better job of readying their students for industry (Whittaker, 2000; Kaner & Padmanabhan, 2007; Jones & Chatmon, 2001).

We studied the difficulties students at a large R1 university faced, while writing unit and integration tests, and connecting them to a continuous integration environment, within a semester-long, industry sponsored course project. Particularly, we studied the difficulties and impediments students in this sophomore-level software engineering course faced in the testing process. We utilized students' own voices to lend nuance and contours to our findings.

2. Related Work and Research Question

Novice approaches to programming errors and debugging received significant attention during the 1980s. Numerous studies have concentrated on novice programmers, considered generally to be those programmers who have received little programming education (a year or less of coursework). Studies related to testing are usually focused on the researcher's or instructor's perspective.

2.1. Software testing - CS1 and CS2

Published studies on the difficulties novice programmers face include: the collection of research papers by Soloway and Spohrer (Soloway & Spohrer, 2013), SmallTalk programming difficulties by Guzdial (Guzdial, 1995), the work by Milne and Rowe (Milne & Rowe, 2002) on CS1 programming difficulties, and a study by Fitzgerald et al. (Fitzgerald et al., 2008) based on a modified replication of Katz and Anderson's quantitative study of novices. Work by Soloway and Spohrer (Soloway & Spohrer, 2013), Milne and Rowe (Milne & Rowe, 2002), Lahtinen et al. (Lahtinen, Ala-Mutka, & Järvinen, 2005), and the Australasian review by de Raadt (de Raadt, 2007), also provide more insights into the difficulties that novice programmers face. These studies showed that understanding of basic concepts and fixing bugs was not as difficult as learning to apply those concepts, and finding bugs.

All of the above mentioned studies focused on programming, and not particularly on software testing. Edwards systematically examined how students test in a course on data structures (Edwards & Shams, 2014). Almost all students only performed 'happy path' testing, testing only the default scenario. Simon et al. (Simon et al., 2008) reported on student responses to a series of four questions designed to identify pre-existing abilities related to debugging and troubleshooting experiences of novice students before they begin programming instruction. Qian and Lehman (Qian & Lehman, 2017) found that novice programmers exhibit various misconceptions and other difficulties in syntactic knowledge, conceptual knowledge, and strategic knowledge. More recently, Bijlsma et al. (Bijlsma, Doorn, Passier, Pootjes, & Stuurman, 2020) investigated how first year students initially view software testing, and what misconceptions they have, through a combination of interviews and exercises. They found that students formulated test cases based on code inspection; many test sets were incomplete; students lacked motivation and time to test; and used incorrect test strategies.

2.2. Software testing - after CS2

There have been fewer studies of intermediate and upper level CS courses regarding testing. Begel and Simon (Begel & Simon, 2008) conducted an observational study of eight recent college graduates in their first six months in software development positions at Microsoft Corporation. They classify some of the common ways new software developers were observed getting stuck: communication, collaboration, technical, cognition, and orientation. Garousi (Garousi, Petersen, & Ozkan, 2016) incorporated "real-world" industrial testing projects in a graduate-level software testing course from 2008-2010, and observed that there was a need to proactively monitor students' progress and provide them with feedback. Hooshangi, Weiss and Cappos (Hooshangi, Weiss, & Cappos, 2015) analyzed the quality of both attack and defense code that students wrote at both graduate and senior undergraduate levels, and found evidence that students who learn to write good defensive programs can write effective attack programs, but the converse is not true.

There is little previous work inquiring into student perspectives on how bug finding and fixing occurs during the software engineering process cycle. As a result, when theories are constructed on how students learn software testing, we do not understand these concepts from a student perspective. There is evidence that testing can be taught, however there are currently no studies investigating student difficulties with software testing in their own voices.

3. Research Question

In this paper we examine how students approached various aspects of software testing (e.g. unit testing, integration testing) and a central practice of DevOps (continuous integration), within the context of a software engineering course with an associated, industry partnered project component. Our singular research question for this study is:

What were the most prevalent difficulties according to students regarding learning unit testing, integration testing and continuous integration?

4. Methods

4.1. Study Context

This research project was determined to be exempt by our University's Institutional Review Board.

Data for this study were collected from five focal participants of a cohort of 38 sophomore students taking a software engineering class in the spring of 2019. Based on guidelines by Creswell and Poth (Creswell & Poth, 2016), between 3-10 participants would typically be sufficient to achieve data saturation for our qualitative interviews.

All students received 3 modules of instruction on unit testing, integration testing, and continuous integration techniques and practices. Peer Instruction (Crouch & Mazur, 2001; Zingaro & Porter, 2015) was utilized in the classroom as the pedagogy of choice along with lectures and comprehensive explanations. Students were assessed on their knowledge through quizzes on each topic (conducted a week after each topic was taught) and a final end-of-semester exam.

Students also worked on a moderately sized greenfield project with requirements provided by a local software company. Student teams of four gathered requirements from the industry sponsor and then designed, implemented, tested and deployed their software using Agile processes using two week sprints. Typically, each team would consist of one UI/UX focused developer, one tester/DevOps student, and two full stack developers. Even though students took on these specialized roles in teams, they all had the same academic preparation in the course. The participants in our study were testers from five different teams.

4.2. System Under Test

The System Under Test (SUT) was a web application that the students designed, developed, tested and deployed on a cloud platform. This application would keep track of employees' past, current and in progress technical and process skills, and would aid the human resource department in the company in identifying employee skills appropriate for future projects. Students were asked to develop a web application that would replace a legacy system that utilized spreadsheet data. The application was required to contain functionality for user authentication, skills classification, job requirements posting, skills selection, and skills matching, and hosted on a cloud platform. The SUT was developed according to the iDesign method, consisting of "accessor" classes for data access,"engine" classes for functional requirement implementation, and "manager" classes for workflow (*iDesign Design Standard Software Design*, n.d.). Students were required to write and run unit tests and integration tests for all methods in all accessors, engines, and managers, with a minimum code coverage of 85%. Students used test doubles such as fakes, mocks, and stubs while creating tests.

4.3. Data Collection

We conducted semi-structured interviews of students (Weiss, 1995; Wolcott, 2005), and combined them with researcher memos, and reflective researcher journals. We maintained and shared field journals wherein we kept jottings (Emerson, Fretz, & Shaw, 2011) that we turned into expanded field-notes. We analyzed these field-notes along with transcribed semi-structured interviews. Notes from student code patterns as well as quiz and exam responses on the topics of unit and integration testing were also included in the analysis.

4.4. Data Analysis

Our analysis of the data employed a parallel approach with reflective collaborative check-ins and theming, consistent with the grounded theory approach in qualitative analysis (Creswell & Poth, 2016). We began with an initial individual coding of transcripts. We generated and assigned over 350 codes. These codes were then mapped into different visual configurations based on code topic groupings and frequency. This was combined with iterative analysis (Anfara Jr, Brown, & Mangione, 2002). Code maps were used to guide code grouping and chunking which supported our collaborative integrative theming and theory development (Corbin & Strauss, 2014). The combination of the two approaches helped us to identify and connect the elements we both noted among the emerging themes.

4.5. Reliability and trustworthiness

To enhance reliability, we focused on intercoder agreement based on the use of multiple coders to analyze transcript data. Two of the authors, both trained in qualitative research methods, analyzed the individual codes separately to come up with themes. We utilized Cohen's Kappa (Hsu & Field, 2003) as a measure of intercoder reliability. We report an intercoder reliability (Creswell & Poth, 2016) of 1.0 (100%) among all themes. To aid in ensuring trustworthiness, students participating in the study were invited to give feedback on our interpretations and theory building. This process is known as member checking (Lincoln & Guba, 1985). Participants were offered a descriptive summary of our themes and theories, and/or a chance to read the entire paper. We invited their open-ended feedback on the extent to which our interpretations and organization resonated with their experiences. Four participants agreed to give written feedback which was used to clarify information and confirm our organization and findings.

4.6. Data Organization

We have worked to forefront the experiences and voices of our participants in the following data sections. Our way of organizing data and making meaning via discussion and interpretation in this study aligns with the way education researchers position participant attrition and participation (Ketelhut & Schifter, 2011; Foley, 2002). Throughout the data presentations that follow, we forefront participant voices in the text and employ student quotations as subheadings .

5. Analysis and Discussion

In this section we write about several of these challenges from our participants utilizing their own voices. We write first about what students said about translating requirements into tests and the impediments they faced along the way; we then present impediments associated with technology-related aspects of learning unit testing, integration testing, and continuous integration. We conclude with the cognitive biases that presented themselves in this process.

5.1. What makes writing and maintaining tests difficult?

"Tests are a lot of work to write. I found it difficult to write just enough. You have to test the satisfactory condition, then the negative condition, then the boundary cases, then the exceptions. It's a lot." - Lisa

"Unit tests keep growing as code changes, I forgot to update them as methods changed." – Adam

A common theme across the students in our study was that they found the complexity involved in developing exhaustive tests daunting. This aligns with the cognitive conflict expressed in previous work (Ginat & Shmalo, 2013; Carver & Risinger, 1987; Qian & Lehman, 2017). Some found test creation, execution, and maintenance time consuming. Keeping up with modifying tests as associated functionality changes and making tests comprehensive in testing all possible code paths was a common issue. Students expressed frustration at "how hard" it was to create and run tests. Test creation was time consuming because of the complexity involved in satisfying several conditions. Test maintenance was difficult because tests had to be checked and modified in alignment with code changes.

"Testing is hard. It takes so much time. It is tempting to just not write tests. . . ." – James

"Testing was rushed. . . ." – Minnie

Writing, executing, automating, and reviewing test cases is an ongoing process that involves making connections between various aspects of software engineering. In an agile software engineering course project, students found themselves having to continuously revisit and possibly modify their tests as the underlying code changed. Our participants expressed frustration and wanted the testing process to end. James' and Minnie's statements above indicate how the pressure and difficulties of testing nearly deterred him from writing them, which was consistent with the findings by Begel and Simon (Begel & Simon, 2008), as well as Bijlsma (Bijlsma et al., 2020).

"They get unwieldy and unmanageable as we keep refactoring our code. Going back and writing helper methods is hard, but the right thing to do." – Lisa

Based on our participant voices, we found four significant factors that could lead to incorrect time estimation due to poor time management. First, novice testers might not know how to divide and prioritize their application code into testable chunks. Second, novice testers might not know application requirements at the granular level needed to create tests. Third, students lacked an awareness of potential difficulties in learning the required technological tools. Fourth, novice testers might not know how to effectively communicate with other team members or with business stakeholders, by asking relevant and timely questions that could help them. Not knowing when and how to ask for help was an issue with new software engineers in the industry as noted by Begel and Simon (Begel & Simon, 2008). Lisa's statement indicated that she understood the idea of writing modular code (helper methods) that are reusable. She also expressed frustration at the continuing process of having to refactor tests as she and her project teammates refactored the underlying code.

5.2. Starting troubles – prioritizing tests, understanding entry points, and creating documentation

"Where do you start testing? What is the most important functionality to test? I had no idea. And I was the designated tester on my team and it didn't help." – Adam

Adam expressed frustration over not knowing where to start testing. Prioritizing test cases based on functionality seemed to be a problem for all of our participants, another unique result we obtained. This is usually achieved by sequencing the testing of software modules that are important from the customer's perspective. Novice testers such as our participants exhibited a lack of understanding on how to make connections between business requirements, previous test cycle experiences on functioning of the existing features, and delivery timelines, which are all essential tasks in prioritizing test cases.

"I had a difficult time understanding whether we needed a text messaging facility along with email or just email? Did we need to include some form of notification?" - Minnie

"It is hard to know whether I have tested the whole thing – have I missed something? I don't know." –Adam

Minnie called attention to her limited knowledge of the application itself. Are certain functionality elements required? Not understanding the requirements means that it was much harder to test those requirements. Adam's concerns about not fully understanding the requirements are similar to Minnie's. Kayla echoed a related but interesting concern. She explained that *"it was difficult to try to find documentation for what needed to be tested; the code almost had no comments in it....".* This lack of code documentation made it difficult for her to not just write tests but to document them.

"Documenting everything as tested it was hard – there was nothing to lead the way, we had to create the documentation as we went along, so I just did not do it." - Lisa

There are often several steps involved in creating documentation for code or tests. Creating a documentation plan, and structuring and designing by using templates or schemas for consistent on-page design are usually the starting point. This has to be followed by creating a simple, logical navigation structure, creating the actual content by starting with a draft, obtaining feedback through peer reviews, and making revisions. It is also critical to maintain the documentation as code changes occur. Without sample documentation to lead the way, as is often the case in greenfield projects, our participants seemed to have difficulty in many of these steps, resulting in unfinished and untested functionality. Prado, Verbeek, Storey and Vincenzi (Prado, Verbeek, Storey, & Vincenzi, 2015) observed that documentation was important to novice testers from a tooling perspective. We concur and expand upon this finding by reporting that our participants found code documentation to be an integral part of the test writing process.

5.3. Determining completeness

"Does my testing accomplish all functionality goals? How do I know for sure?" – Adam

"Code coverage is not everything when we test, but it is hard to know whether you tested everything." –

James

Our novice software engineers often had trouble determining when to stop testing. Establishing clear and fine-grained exit criteria for an application is not always easy. Completeness of testing is usually decided based upon metrics such as number of test cases executed, passed, failed, and code coverage. Identifying and testing high priority bugs is a critical step in ensuring test completeness, as is examining code coverage in combination with a successful execution of major functional or business flows. Our participants struggled to understand and connect these concepts and apply them to the framework of writing and executing tests, resulting in a lack of well established critical test cases.

Participants found it hard to understand and compartmentalize code into chunks that could be unit test, consistent with the findings by Milne and Rowe (Milne & Rowe, 2002). They found the lack of documentation to be a notable impediment. Our participants struggled with: i) identification of which code to test ii)isolation of the unit under test iii) when to determine completeness of the testing process and iv) understanding code coverage and its relationship to testing. These results concur with and expand upon the findings of Daka and Fraser (Daka & Fraser, 2014), who revealed in their study that (i) and (ii) were the two most difficult aspects related to writing new tests. Unit testing may still be underestimated by testers due its apparent simplicity compared to other levels of test.

5.4. Requirements and testing

"Requirements need to be honed down to unit test specifics." - James

James had a novel take on refining requirements. He felt that the documented requirements were not granular enough for him to write tests. However, unit tests are typically written to test units of code, not requirements. James' statement shows a preference for a test-first code paradigm similar to Test Driven Development, where requirements are typically translated to tests before they get implemented as code.

"There was a definite mismatch with the code and requirements, and I missed out on several bugs because not all negative flows could be inferred from the requirements." - Lisa

Lisa was able to connect requirements to code, and subsequently to tests. However, fine grained test case writing, involving recognizing various program flow directions, was a struggle. In spite of that, it is noteworthy that at a novice level, she was able to recognize this issue and be aware of the incompleteness of her test suite. It is also indicative of her nuanced grasp of testing beyond just the 'happy path' or expected behavior, as evidenced by her comment on 'negative flows'. This is consistent with the findings by Edwards (Edwards & Shams, 2014).

Unit and integration tests are meant to test code, not whether the code implemented requirements correctly. It seems that our participants had a strong inclination to rely on requirement documents and stakeholder provided documentation to jump-start their unit testing process. An issue that could arise with relying heavily on requirements, instead of relying on implemented code, is that all documented requirements may not be implemented. On the other hand, an awareness of application requirements could help them discover bugs quicker, when they see a mismatch between requirements and implemented code.

5.5. Difficulty using tools

"CI servers are hard work. Setting it up, learning yaml, and getting those dlls to behave wasn't easy." – Kayla "Wading through code coverage was hard; and code coverage doesn't even tell me if my program is tested completely." - James "Just learning to set up a mock for our database took me a long time. . . " – Minnie

"Visual Studio's debugger helped me step through code – initially it took me a while to learn how to set breakpoints etc. because I had never done it before. . . " - Kayla

All our participants indicated some sort of difficulty with tools involved with writing and integrating tests with a continuous integration environment. Code coverage tools seemed to pose a challenge as well, as evidenced by James' comment. Creating mock objects during integration testing was another area of difficulty. Tools related to continuous integration usage of the Visual Studio debugger, along with learning to set breakpoints was a time-consuming process. These are in alignment with the findings by Begel and Simon (Begel & Simon, 2008) regarding new hires in the software industry.

5.6. Communication issues and difficulties due to late stage testing

"I had to try to trace back to where the accounting error was happening – all the student loan calculations were off by a couple of points- how did this happen? It took me forever to figure it out, and even then, I had trouble reproducing the error, and it was all on me." – Kayla

Kayla's frustration highlights an overall unsaid issue – communication within the team. Is the description well-written? Can the programmer understand what was wrong or why the tester thought something was wrong? Does existing documentation include steps to reproduce the problem, even basic information about what they were doing when the problem happened? In this particular case, Kayla seemed to have no help. She did not know how to ask for help and what to ask. This is a common problem observed with novice programmers (Simon et al., 2008).

Early testing, something favored by Bijlsma (Bijlsma et al., 2020), was also seen as a key, often missing piece. Lisa said, *"we wrote all the code and then started testing, but I got to know later that some other teams started testing as they were coding. I wasn't sure what the sponsors wanted in terms of tests, and it was too late to ask."* Minnie also reflected on the need to test earlier saying, *"we started testing very late; we just wrote unit tests for the most part and skipped many of the required integration tests which I think would have been beneficial."*

Thus, late stage testing was a problem, which was indicative of not knowing how and when to ask questions of their fellow students, instructor, TAs, and the industry partner. This aligns with Begel and Simon's findings (Begel & Simon, 2008) in that our participants did not ask questions soon enough nor did they know how to ask questions at the appropriate level.

5.7. Cognitive Biases

Cognitive biases are defined as the deviation of the human mind from the laws of logic and accuracy (Stacy & MacMillan, 1995). A cognitive bias is a pattern of deviation in judgment that occurs in specific situations, and leads to perceptual distortion, inaccurate judgment, illogical interpretation, or what is broadly called irrationality. Confirmation bias is an important cognitive bias and is defined as the tendency of people to seek for evidence that could verify their hypotheses rather than refuting them (Calikli & Bener, 2010). Congruence bias, which is a special case of confirmation bias, is the tendency to over-rely on testing one's initial hypothesis while neglecting to test alternative hypotheses. It manifests in software testing as a 'positive test bias' (Mohanani, Salman, Turhan, Rodríguez, & Ralph, 2018).

"Writing integration tests was mainly to check that the database worked correctly." – Minnie

"Integration tests ensured that the UI and database behaved as expected." – James

"Mock objects in integration tests helped verify database connections worked correctly... but that was all I did for integration tests." – Adam

We found that 3 of our 5 participants exhibited congruence bias in integration testing but not in unit testing – preferring the 'happy path', or expected behavior, as the only path to test. This is in alignment with the findings by Bijlsma (Bijlsma et al., 2020). We reason that this could be since integration testing was a new concept to them, compared to unit testing, which students had studied in a previous course.

We also found a lack of awareness regarding domain knowledge and the presence or absence of run time errors, which could help moderate this bias (Salman, 2016). None of the CS major participants (with previous coding internship experience) mentioned run time errors or domain knowledge in their interviews. The CS major participants with programming expertise did exhibit congruence bias, and this leads us to concur with Salman (Salman, 2016) that programming expertise seems to have no effect regarding who exhibits congruence bias.

Another cognitive bias we encountered among our participants was availability bias, which is the tendency for easy-to-recall information to unduly influence preconceptions or judgments (Mohanani et al., 2018). Our participants expressed frustration based on the lack of documentation (Section IVA), and showed that they ended up selecting test cases based on not being able to perform exhaustive testing, but on mental shortcuts that tend to rely on anecdotal examples.

"Seeing this error in prod once made me want to test it thoroughly in every engine and accessor." – James

James' comment indicates an availability bias. Just because an error showed up in production once does not mean it needs to be tested in every layer of design, regardless of its relevancy. Surprisingly, representativeness (dropping error-producing test cases, misrepresenting code features) was not an issue for our participants in this study, however tedious the test-writing process was.

5.8. Implications for teaching and threats to validity

As with any empirical study, there are various internal and external threats to validity that our results are subject to. Our sampling method included 5 interviews (2 women, 3 men) that yielded rich insights and followed recommended guidelines for saturation on the topic studied (Creswell & Poth, 2016). While our analysis was systematic, other researchers may glean different themes, attributes and definitions than ours from the same raw data. As we interviewed students from a single cohort, with identical prior coursework, we deem our findings to be valuable and relevant within the context of our study.

Our participants voiced concerns about the disconnect between requirements and testing, and difficulty in developing product knowledge enough to test adequately, one of the results unique to our study. Teaching students how to trace code and test cases to requirements and emphasizing activities that help them identify business priorities may prove helpful with the above impediments. Teaching students to create and groom product backlogs that prioritize business tasks and technical tasks based on client priorities may help mitigate the difficulties they expressed regarding prioritizing tests, understanding entry points, and creating documentation.

Determining completeness of the test suite beyond code coverage was another significant and unique challenge that our study uncovered. Most of the communication issues that our participants expressed were caused by late stage testing, when in fact, beginning incremental testing at earlier stages could have been helpful. We propose that a heightened focus on understanding business and technical priorities, as well as more instruction and practice on technical tools (such as IDEs and CI servers) can help mitigate the above mentioned impediments. Such practice may be provided in students with a large pre-existing code base to which they must fix bugs (injected or real) and write additional features (Begel & Simon, 2008).

6. Conclusion and future work

Software testing is both art and science. In answering our research question, "What were the most prevalent difficulties according to students regarding learning unit testing, integration testing and continuous integration?", we found that our novice testers were plagued by a plethora of problems. The most notable ones were: communication within the team and other stakeholders, prioritization of features to be tested, entry and exit criteria for tests, difficulties with learning the tools associated with writing and executing tests, the sheer time commitment involved in designing, writing and implementing meaningful tests, not knowing what kind of questions to ask and of whom, how to look for test completeness beyond code coverage, and the parallel temporal nature of code base development and testing. In addition, we explored the cognitive biases that proved to be impediments to these testers in accomplishing their testing goals without frustration. However the robustness our participants described in testing suggests it was an impediment to student learning, potentially souring them on the testing process. We also outlined potential mitigation strategies for some of these difficulties and biases.

Helping students recognize and identify these challenges in their own voice, as we have done in this exploratory pilot study, is a first step. Each one of these impediments warrants a more detailed exploration. In future work we intend to focus on challenges specific to integration testing and continuous integration in greater depth, as well as the role of cognitive biases in learning those topics. In particular, we intend to focus on the relationship between the process of learning software testing and the process of gathering and managing requirements in tandem, using the Agile methodology. We also intend to explore the effectiveness of other mitigation strategies we outlined above, in reducing student frustration and enhancing the testing and DevOps learning experience.

7. References

Anfara Jr, V. A., Brown, K. M., & Mangione, T. L. (2002). Qualitative analysis on stage: Making the research process more public. *Educational researcher*, *31*(7), 28–38.

Begel, A., & Simon, B. (2008). Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research* (p. 3–14). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/1404520.1404522` doi: 10.1145/1404520.1404522

Bijlsma, A., Doorn, N. N., Passier, H., Pootjes, H., & Stuurman, S. (2020). How do students test software units?: Part one: Their natural attitude diagnosed.

Calikli, G., & Bener, A. (2010). Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proceedings of the 6th international conference on predictive models in software engineering* (pp. 1–11).

Carver, M. S., & Risinger, S. C. (1987). Improving children's debugging skills. In *Empirical studies of programmers: Second workshop* (pp. 147–171).

Clark, N. (2004). Peer testing in software engineering projects.

Corbin, J., & Strauss, A. (2014). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications.

Creswell, J. W., & Poth, C. N. (2016). *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications.

Crouch, C. H., & Mazur, E. (2001). Peer instruction: Ten years of experience and results. *American journal of physics*, *69*(9), 970–977.

Daka, E., & Fraser, G. (2014). A survey on unit testing practices and problems. In *2014 ieee 25th international symposium on software reliability engineering* (pp. 201–211).

de Raadt, M. (2007). A review of australasian investigations into problem solving and the novice programmer. *Computer Science Education*, *17*(3), 201–213.

Edwards, S. H., & Shams, Z. (2014). Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 171–176).

Emerson, R. M., Fretz, R. I., & Shaw, L. L. (2011). *Writing ethnographic fieldnotes*. University of Chicago Press.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116.

Foley, D. E. (2002). Critical ethnography: The reflexive turn. *International Journal of Qualitative Studies in Education*, *15*(4), 469–490.

Garousi, V., Petersen, K., & Ozkan, B. (2016). Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, *79*, 106–127.

Ginat, D., & Shmalo, R. (2013). Constructive use of errors in teaching cs1. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 353–358).

Guzdial, M. (1995, 01). Centralized mindset: a student problem with object-oriented programming. In (Vol. 27, p. 182-185).

Hooshangi, S., Weiss, R., & Cappos, J. (2015). Can the security mindset make students better testers? In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 404–409).

Hsu, L. M., & Field, R. (2003). Interrater agreement measures: Comments on kappan, cohen's kappa, scott's π, and aickin's α. *Understanding Statistics*, *2*(3), 205–219.

iDesign Design Standard software design. (n.d.). `https://idesign.net`. (Accessed: 2021-01-03)

Jones, E. L., & Chatmon, C. L. (2001). A perspective on teaching software testing. *Journal of Computing Sciences in Colleges*, *16*(3), 92–100.

Kaner, C., & Padmanabhan, S. (2007). Practice and transfer of learning in the teaching of software testing. In *20th conference on software engineering education & training (cseet'07)* (pp. 157–166).

Ketelhut, D. J., & Schifter, C. C. (2011). Teachers and game-based learning: Improving understanding of how to increase efficacy of adoption. *Computers & Education*, *56*(2), 539–546.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *Acm sigcse bulletin*, *37*(3), 14–18.

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry. newberry park.* Ca: Sage.

Meneely, A., & Lucidi, S. (2013). Vulnerability of the day: Concrete demonstrations for software engineering undergraduates. In *2013 35th international conference on software engineering (icse)* (pp. 1154–1157).

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies*, *7*(1), 55–66.

Mohanani, R., Salman, I., Turhan, B., Rodríguez, P., & Ralph, P. (2018). Cognitive biases in software engineering: a systematic mapping study. *IEEE Transactions on Software Engineering*.

Osterweil, L. (1996). Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, *28*(4), 738–750.

Prado, M. P., Verbeek, E., Storey, M.-A., & Vincenzi, A. M. (2015). Wap: Cognitive aspects in unit testing: The hunting game and the hunter's perspective. In *2015 ieee 26th international symposium on software reliability engineering (issre)* (pp. 387–392).

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1–24.

Salman, I. (2016). Cognitive biases in software quality and testing. In *2016 ieee/acm 38th international conference on software engineering companion (icse-c)* (pp. 823–826).

Simon, B., Bouvier, D., Chen, T.-Y., Lewandowski, G., McCartney, R., & Sanders, K. (2008). Common sense computing (episode 4): Debugging. *Computer Science Education*, *18*(2), 117–133.

Smith, J., Tessler, J., Kramer, E., & Lin, C. (2012). Using peer review to teach software testing. In *Proceedings of the ninth annual international conference on international computing education research* (pp. 93–98).

Soloway, E., & Spohrer, J. C. (2013). *Studying the novice programmer.* Psychology Press.

Stacy, W., & MacMillan, J. (1995). Cognitive bias in software engineering. *Communications of the ACM*, *38*(6), 57–63.

Weiss, R. S. (1995). *Learning from strangers: The art and method of qualitative interview studies.* Simon and Schuster.

Whittaker, J. A. (2000). What is software testing? and why is it so hard? *IEEE software*, *17*(1), 70–79.

Wolcott, H. (2005). *The art of fieldwork. walnut creek, calif.* AltaMira Press/A Division of Rowman & Littlefield Publishers, Inc.

Wong, W. E., Bertolino, A., Debroy, V., Mathur, A., Offutt, J., & Vouk, M. (2011). Teaching software testing: Experiences, lessons learned and the path forward. In *2011 24th ieee-cs conference on software engineering education and training (csee&t)* (pp. 530–534).

Zingaro, D., & Porter, L. (2015). Tracking student learning from class to exam using isomorphic questions. In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 356–361).