The impact of POGIL-like learning on student understanding of software testing and DevOps: A qualitative study

Bhuvana Gopal
School of Computing
University of Nebraska-Lincoln
bhuvana.gopal@unl.edu

Ryan Bockmon
School of Computing
University of Nebraska-Lincoln
ryan.bockmon@huskers.unl.edu

Stephen Cooper
School of Computing
University of Nebraska-Lincoln
stephen.cooper@unl.edu

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

Abstract

In this study, we analyze students' understanding of unit testing, integration testing and continuous integration in a semester long undergraduate software engineering course, after they underwent instruction using POGIL-like, a guided inquiry based pedagogy. At the end of the course, we collected student responses to open ended questions regarding their understanding of these topics, combining them with researcher memos as well as reflective researcher journals. We analyzed these written responses and identified the themes that we came up with regarding how students learned and potentially overcame difficulties with software testing and DevOps. Some of those themes that emerged from our qualitative analysis were: What makes writing and maintaining tests difficult? Where do you start unit testing a method? How do you know if you have written enough tests for the System Under Test (SUT)? How do you identify the most important functionality to test, in a SUT? Does your testing accomplish all functionality goals?

We investigate and discuss students' answers to these questions in detail. We attempt to understand if the POGIL-like approach helped students overcome some of the difficulties students expressed in our earlier work on the same topic in a previously published study.

1. Introduction

Teaching software testing is an important part of teaching software engineering. How software is tested heavily impacts how reliable the code is (Lemos, Ferrari, Silveira, & Garcia, 2015). There is often a disconnect in how testing practices are taught in undergraduate software engineering education (S. Edwards, 2004), often with little or no emphasis on real practical training (S. Edwards, 2004; Bijlsma, Passier, Pootjes, Stuurman, & Doorn, 2020). How well do students know software testing? This is often overlooked because students often find learning software testing challenging, and assessing the extent of their learning is difficult (Gopal, Cooper, Olmanson, & Bockmon, 2021).

In this paper, we attempt to discover what students learned in the topics of unit testing, integration testing and continuous integration (CI) in an undergraduate, predominantly sophomore/junior level software engineering course. Students were instructed using a collaborative pedagogy called POGIL-like (Gopal & Cooper, 2022) which is an implementation of Process Oriented Guided Inquiry Based Learning (POGIL) (Yadav, Kussmaul, Mayfield, & Hu, 2019). "POGIL" is a copyrighted term and we use "POGIL-like" to indicate our pedagogy throughout this paper. For this study, we asked students to express in their own words their understanding of the topics. We conducted a qualitative analysis of the rich set of data we obtained from student reflections of their learning, and uncovered four themes, on which we elaborate in this paper. In delineating these themes we expand on how students overcame various difficulties in learning the topics.

1.1. Overview of POGIL-like

POGIL-like is a pedagogy where students are organized into small teams. Students collaborate and actively learn to work together to co-construct knowledge on the topic being taught, using the idea of concept invention (Kussmaul, 2011). Students start each class session with little or no prior knowledge of the topic, so they can benefit from the co-construction of knowledge through POGIL-like activities without added misconceptions.The instructor serves as an active facilitator, walking around the classroom and helping students as needed during the session. Each team consists of 4-6 students, and each student has a specific role to play. There are 4 roles: Manager, Recorder, Presenter and Reflector. Students engage with "models" and "activities". Models are a compilation of content knowledge that the students need to master. Models typically contain figures, tables, equations, and code snippets in addition to plain text. Activities contain critical thinking questions and hands-on exercises. An overview of the POGIL-like pedagogy and its salient features can be found in our previous work (Gopal & Cooper, 2022).

The student designated as the Manager keeps track of time and keeps everyone in the group focused throughout the session. The Recorder jots down everything that is being discussed during the activities. The Reflector helps the team conduct a mini retrospective after each activity and reflect on what worked well and what did not. After working through the models and activities, the student designated as the Presenter provides a summary of what the team learned during the session, and the instructor discusses the findings with the whole class. The way that students build their knowledge in POGIL-like is by exploring the models to "invent" important concepts and eventually apply what they learned (Hanson, 2005) through the co-operative, role-based interactions that they have with fellow students within their small groups. This knowledge exploring, concept inventing and application process can be utilized for both technical content and process-related content (e.g. problem solving, teamwork, and written/oral communication) (Hu, Kussmaul, Knaeble, Mayfield, & Yadav, 2016).

The rest of the paper is organized as follows. We present prior work in software testing, POGIL-like, and student reflections in Section 2. We present our research question in Section 3. Our research methods are explained in Section 4. Themes from our data analysis along with a discussion are presented in Section 5. We detail the threats to the validity of our study in Section 6, and conclude in Section 7.

2. Prior Work

Software testers in the industry often lack adequate academic preparation (Buffardi & Edwards, 2014; Wong et al., 2011; Garousi & Zhi, 2013; Chen, Zhang, & Luo, 2011; Ng, Murnane, Reed, Grant, & Chen, 2004). Several studies have been conducted in software testing education, with different research questions (Clark, 2004; Elbaum, Person, Dokulil, & Jorde, 2007; Clarke, Pava, Davis, Hernandez, & King, 2012; S. H. Edwards & Shams, 2014). Some studies have focused on challenges with software testing (Drake & Drake, 2003; Aniche, Hermans, & Deursen, 2019; Greising, Bartel, & Hagel, 2018) and explored ways to teach testing and DevOps. All these studies explored different ways to teach software testing. There are fewer studies on the impact of a specific teaching approach on how well students learned the testing topic.

Taipale and Smolander (Taipale & Smolander, 2006) found that communication efficiency mattered, and that early, risk based testing was important. They also emphasized that testing needed to be specific and tailored to business needs. Memar et al. (Memar, Krishna, McMeekin, & Tan, 2018) qualitatively studied students' evaluations of a gamified approach to teaching software testing, emphasizing the importance of feedback. Kennedy and Kraemer (Kennedy & Kraemer, 2019) studied what students' thoughts were when they were asked to "Please, think aloud" as they developed code. They analyzed video and audio recordings capturing students' thoughts in real time, and found that students showed uncertainty regardless of success at task completion.

The qualitative study by Florea and Raluca (Florea & Stray, 2020) asked experienced software testing professionals in industry what was important for software testers in terms of background, skills, learning preferences, and role profiles. They found most of their participants preferred exploratory testing. Curiosity was the most valued quality in a tester. An interesting conclusion they came to was that software testing skills needed were largely undefined, unclassified and unorganized, and increased with each new task. Most participants learned testing on the job, informally, and felt the need for better approaches to teaching testing in post-secondary education. Their study strongly emphasized the need for educators to think outside the box of traditional methods when it comes to teaching students software testing.

Stray et al (Stray, Florea, & Paruch, 2021) qualitatively studied the human factors of Agile software testers. They found strong software testers had: the ability to see the whole picture, good communication skills, detail-orientation, structuredness, creativeness, curiosity, and adaptability. They proposed that these seven qualities be taken into consideration when organizations recruit testers for agile software.

In our previous work (Gopal et al., 2021) we conducted a qualitative study using semi-structured interviews and identified various difficulties that plagued our novice testers 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. In another study, we studied quantitatively how students answered questions on unit testing, integration testing and CI, in pre- and post-tests, after being taught using the POGIL-like approach, and found that there were statistically significant raises in student scores in the POGIL-like group compared to a pure lecture based group (Gopal & Cooper, 2022).

As we can see from the studies above, several interesting aspects of teaching and learning software testing have been explored. There is not much literature on how students express their understanding of software testing and DevOps topics, especially in the context of specific teaching and pedagogical approaches.

3. Research Question

In this paper we examine how students understood the broad topics of software testing and DevOps, within the context of a POGIL-like software engineering course. Our research question for this study was:

RQ: To what extent did undergraduate students learn and overcome known difficulties with software testing and DevOps when instructed using a POGIL-like pedagogy?

4. Methods

4.1. POGIL-like: Student and instructor roles

Our student teams consisted of 4 distinct roles (Hu & Shepherd, 2014): Manager, Recorder, Presenter and Reflector. These roles were assigned to each student to foster interdependence as well as individual responsibility and accountability to the success of the team (Kussmaul, 2012).

In our classroom the instructor was able to specifically assign groups and observe closely how students interacted with each other using the prescribed roles (Hu & Shepherd, 2013). The instructor focused on helping students develop process skills, specifically, problem solving, teamwork and critical thinking (Hu & Shepherd, 2013). The instructor offered additional guidance as students worked in teams (Hanson, 2005; Kussmaul, 2011).

4.2. POGIL-like Activity development: Models, E-I-A cycles and D/C/V questions

A model in a POGIL-like activity denotes the content that we would provide for students to know, during lectures or required readings (Maher, Latulipe, Lipford, & Rorrer, 2015), but presented with action verbs, figures, pictures and tables as needed, instead of copious quantities of plain flowing text. We used a combination of three types of questions -Directed (D), Convergent (C) and Divergent (V) questions (Gopal & Cooper, 2022). A POGIL-like learning cycle employs a series of Explore-Invent-Apply (E-I-A) activities comprised of D/C/V questions.

4.3. Study Context

This research project was determined to be exempt by our University's Institutional Review Board. Data for this study were collected from 22 participants of a cohort of 62 sophomore/junior/senior students taking a software engineering class in the Fall of 2021. All students were taught using POGIL-like on unit testing, integration testing, and continuous integration. 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.

4.4. Data Collection

We presented students with an online questionnaire and combined them with researcher real-time memos, and reflective researcher journals. At the end of the semester, we surveyed the students an open ended questionnaire where we asked students to describe what they learned from the POGIL-like sessions in software testing and DevOps. We used simple, non-leading prompts such as "Describe what you understood about unit testing", "What were some specific things you learned about integration testing", and "What are the main features of continuous integration?".

We created and maintained reflective journals based on our real time field notes, observing students during the POGIL-like exercises (Emerson, Fretz, & Shaw, 2011). We analyzed students' written responses to the open ended questionnaire along with these field-notes. We corroborated our findings with our notes on student code patterns and quiz performance on the topics of unit and integration testing to lend support to our findings.

4.5. Content Analysis

In analyzing our data, we used a parallel approach with reflective collaborative check-ins and content analysis (Hsieh & Shannon, 2005) combined with 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 500 codes. We grouped the codes into meaningful chunks and counted the frequency of each code/code group. We performed this analysis iteratively and mapped these codes into code maps, which are essentially visual layouts (Anfara Jr, Brown, & Mangione, 2002). We used the code maps to help the process of code grouping and chunking. Finally, we developed our theory based on integrating these code groups with our journals and field notes (Corbin & Strauss, 2014). Two independent coders worked on coming up with overarching themes guided by participants' own voices. This approach helped us to identify and connect the elements we both noted among the emerging themes.

4.6. Reliability

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 presented through students' responses. 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.

4.7. Data Organization

In the following data sections, our aim is to highlight and bring to the forefront, the voices and experiences of our participants. We have followed existing research guidelines on how to position student participation, and our method for meaning making involves understanding recurring expressions of participant sentiments and ideas by taking into account the context in which they were written and submitted (Ketelhut & Schifter, 2011; Foley, 2002). We elucidate a coherent set of data presentations in the following section, highlighting and bringing to the forefront the voices of participants through their written submissions, and utilizing students' own words in the subheadings. We have anonymized the names of students and used pseudonyms instead. In this study, our focus was to see if students had gained any further understanding on the content topics through POGIL-like, and we used our earlier pilot study (Gopal et al., 2021) to inform us of the difficulties that students faced while learning testing and DevOps. Utilizing POGIL-like, did students learn to overcome the difficulties we uncovered earlier? This is the primary focus of our analysis.

5. Analysis and Discussion

In this section we write about several thoughts and impressions of software testing and DevOps, from our participants, employing their own words. We first elaborate what students felt, within the context of learning with POGIL-like, what made learning testing difficult. Next, we present how they determined entry points into the System Under Test (SUT), followed by how they would identify the most important functionality to test. We then present how they determined when to stop testing, and determine test completeness. We conclude with their understanding of whether testing relates to requirements or not.

5.1. Theme 1: What makes writing and maintaining tests and CI pipelines difficult?

"Simply put, change. Code changes, expected behavior requirements change, the framework you work on may have even changed. Keeping up with unit tests, integration tests and maintaining regression testing can cumbersome over time." - Alice.

The fundamental difficulty in testing was the volatility of the entire system and its environment. Alice honed in on this very important problem and displayed an understanding of testing needs changing as a result of changing requirements, and hence code. She very elegantly summarized all the subsystems that could be involved in a cascade of changes when requirements change.

"I think the hardest part about writing tests is to account for all possibilities of regular output and for possible edge cases that could take place.....Testing for edge cases will just happen as you test your application. It makes it hard to maintain tests because as new features are added you will have to adapt your tests based off that." - Ben.

Ben brought a different difficulty to the forefront - edge cases. Edge test cases describe possible but unknown scenarios - which could very well be present at any stage of development. Testers need to be on a constant lookout for edge cases. Boundary testing is a useful technique to find edge cases, specifically with extreme input values. Students often find testing for edge cases difficult since it is hard to know all edge cases that can happen. To know where the boundary conditions are, one needs to know where the non boundary conditions are- where the normal test cases lie. Ben's statements also indicate that the ever changing nature of the codebase made it difficult to keep up with edge test cases.

It is interesting to note that while we uncovered two distinct difficulties with learning testing in our analysis above, the other difficulties we discovered in our previous study (Gopal et al., 2021) were absent.

5.2. Theme 2: Entry and exit points, anatomy of a unit test, integration testing, test doubles, setting up CI, new bugs

"Start with the least complicated portions of the code first and then the most complicated ones." - Lisa.

Lisa's approach to starting where to test involved a recognition of what was complicated in the method. This shows that the tester needs to have an overall picture of what the method is trying to accomplish and what parts of it are complicated.

"When I am testing a method, I start by testing to make sure it is receiving the correct inputs. This means the parameters are passed in properly, and the method receives the necessary data." - Matt.

Matt's approach focused on the inputs and parameters required to call the method correctly. This is a good way to start, without necessarily knowing what the actual complexity of the method is.

"Then, I test the method by running it locally." - Matt.

Matt explains that the method is run "locally" - leading us to believe that he meant that the method is called inside the test method's test context, inside a unit test.

"To test a method by Unit testing, there are three steps: Arrange, Act, and Assert. First you arrange the test and set up everything necessary for it. Second you act, which are the steps taken needed to do the testing of the method. Lastly you assert, which is to look at the outcome and check if it is what's expected." - Gabe.

"I begin by creating a test class to contain my test cases. In each test case, I call the method directly. This includes arranging, acting, and asserting. I finish by running all my test cases through the Test tab in Visual Studio and making sure each one passes." - John.

"We use the process of Arrange, Act, Assert. First, we must arrange, or set up and initialize a method to be tested. Secondly, we act upon that method, which means just executing it. Lastly, we will assert, or return a pass or fail value to show if we have passed or have failed the test." - Molly.

Several students, including Gabe, John, and Molly utilized the Arrange-Act-Assert concept to start testing. This was an intended learning outcome for both unit and integration testing modules, and unlike prior instruction using lecture or peer instruction (Gopal et al., 2021), students in this study seemed to have a better understanding of how to start testing a method.

"All components that will be tested in integration test should be already have unit tests." - Tia.

"Then, I integrate it with the rest of the system and test it there. This is important because it has to function properly with the other methods in the system, otherwise changes have to be made to allow them to integrate properly." - Matt.

With the above statements, we see that Matt and Tia had an understanding of integration testing in combination with unit testing. Testing a method does not simply mean that it works correctly within the test context in an isolated fashion, but it also means that the method works with the other methods it interacts with, other subsystems it depends on, and integrates seamlessly with the entire SUT.

"The unit tests should cover the workflows that the method will handle. I would generally start with a valid case, followed by broken data, whether it is malformed data, no data, etc. I begin by creating a test class to contain my test cases. In each test case, I call the method directly. This includes arranging, acting, and asserting. I finish by running all my test cases through the Test tab in Visual Studio and making sure each one passes." - Pranav.

Pranav's explanation of how to start testing a method demonstrates an understanding of several key concepts in how to start testing: workflows, valid inputs, broken/malformed/lack of data, the arrange-act-assert paradigm, and the tool support needed to run tests in a .NET environment (which is the technology stack the students learned).

"When bugs arises, add a test case for that bug, and modify the method to resolve that bug. Start with unit tests. Make sure to mock out any inputs that you expect and any dependencies. " - Sam.

"When testing a method, I would first see if there are any obvious bugs like missing syntax. Have variables properly labeled. Have variables initialized and check for null values. Then create a Unit Test using a unit test framework to test each method in isolation. The Unit Test should follow an arrange, act, and assert pattern. Create a test following the pattern with arrange is to set up the objects or the environment for the testing, the act is to call the function, and assert is to verify if the actual output is with the expected output using the assert collection. Then run the test and analyze any test that fails so that the problem can be rectified." - Macy.

Sam and Macy focused on how we unit test bugs. Macy echoed the arrange-act-assert pattern as being vital to implementing a unit test. Sam explained that mocking dependencies is important. These students displayed a deep understanding of how the testing process works, accounting for new bugs that arise in the code, and using test doubles when needed. This is in direct contrast to the lack of understanding of test doubles and new bugs they displayed in our earlier work (Gopal et al., 2021).

5.3. Theme 3: Prioritizing test functionality and what to continuously integrate

In the following subsections we focus on what our participants told us regarding how they identified the most important functionality to test, and how they determined when they had tested enough.

"First, understand what the system is supposed to accomplish. I believe the most important functionality to test would be to test the logic or computational functions. If the computational functions are not

correct then the system would produce erroneous output. The entire purpose of any program is to have well-defined business rules implemented through logic to produce the correct output." - Alex.

"I'd recommended to look over the project documentation . If we see which part of the project being influenced and used by a lot of methods , we will prioritize testing it." - Kara

Alex and Kara explained in detail how they identified the most important functionality to test. Particularly noteworthy is his understanding of business rules and requirements being closely related to testing. Referring to documentation is another technique, but sometimes this might not be feasible, especially for greenfield projects.

"Another way to identify the most important functionality is to see which methods get called often." - Alex.

Frequency mapping of method calls is one of the methods that is commonly used in the software industry to see which methods need to be tested first, and as a novice tester, Alex seemed to have obtained a grasp of this useful technique. This could also be due to a prior industry internship, and not just an effect of the classroom instruction he underwent using POGIL-like.

5.4. Theme 4: Test completeness beyond code coverage

"Testing has accomplished all functionality goals when we have detected all known bugs and can prevent them. When the clients and stakeholders are satisfied with the product. When sensitive data is protected and tested in transit and rest." - Daniel.

"For more complex, abstract code you may never catch every test case, so you may never know for 100% certainty if it will function as intended. Even though you can try to cover every possible way to cover you code, there may just be one small input, or parameter that will break it, that no one accounted a test case for." - Abe.

Daniel and Abe displayed a good understanding of how test completeness can be determined. Particularly noteworthy is that beyond having enough tests to cover all possible scenarios, Daniel mentioned stakeholder satisfaction and sensitive data testing was accomplished, acknowledging that sensitive data could be at rest or in transit and need to be tested in both scenarios. Abe's acknowledgement that it is virtually impossible to completely test a complex system displays a nuanced understanding of the testing process.

6. Implications for teaching and threats to validity

Based on our data analysis, we found that students did not focus on DevOps enough. They understood various concepts behind and application scenarios of unit and integration testing, but were notably silent on continuous integration, its importance or benefits.

In any qualitative study, validity is expressed in terms of researcher bias, reactivity and respondent bias (Lincoln & Guba, 2006). As for researcher reflexivity, the primary author of this paper acknowledges that her extensive software industry background positions her to interpret the data with a decidedly industry focused bent. To mitigate the effects of these aspects, we followed some of the suggestions by Robson (Robson, 2002). We engaged with our students throughout the semester, and did not just ask them for responses on the written questionnaire. Students whose responses we used had all given their informed consent to the study. Students' written answers were a reflection of our semester-long involvement with them. We triangulated the data we obtained from their written responses with their quiz performances and our audit trail, to minimize the chances of students simply copying what they think is the right answer. We kept the entire process of data collection, analysis and dissemination transparent to the students, and tried to eliminate any potential "people-pleasing" answers.

We also followed recommended guidelines for saturation on the topic studied (Creswell & Poth, 2016). However, even with our systematic analysis, it is possible that other researchers may distill different themes and ideas than ours from the same raw data. Since we triangulated our data from 22 student responses from a single cohort with the same instructor and instructional pedagogy with the same topics of instruction, we deem our findings to be valuable and relevant within the context of our study.

In our previous study (Gopal et al., 2021) we found several difficulties that students faced while learning software testing and DevOps. The pedagogy of choice in that study was peer instruction (Mazur, 1997). With the same content, same instructor, and similar prior academic preparation, but with the instructional mode being POGIL-like, our participants seemed to have overcome many of the difficulties expressed previously.

Students in our study exhibited an understanding of how to start testing a method. They understood the anatomy of a unit test, and how test doubles could be used in both unit and integration testing. They realized that test completeness went beyond code coverage metrics, and recognized the importance of edge cases and boundary conditions. In our previous study, most of the communication issues that our participants expressed were caused by late stage testing - in this study, our students, started testing early, and tested often, which resulted in no significant mentions of communication issues.

7. Conclusion and future work

In a nutshell, understanding business priorities, starting to test early and testing often, knowing that test completeness depends on stakeholder agreement, prioritizing tests, understanding entry points, recognizing the importance of integration testing, and relying on existing documentation are the major themes that POGIL-like instruction, with its emphasis on process and activities seems to have achieved on our students.

In answering our research question, "To what extent did undergraduate students understand software testing and DevOps when instructed using a POGIL-like pedagogy?", we conclude that POGIL-like is an effective way to teach unit testing, integration testing and continuous integration. POGIL-like helped students gain understanding beyond the surface level, well into the higher layers of Bloom's taxonomy (Bloom, 1984). We surmise that the heavily process oriented nature of the POGIL-like pedagogy, with its E-I-A cycles and D/C/V questions, forced students to think deeper, and go beyond a surface understanding of the "How" to involve more of the "Why?".

In future work we intend to focus on student understanding of CI, and its role in enhancing transparency and visibility to the stakeholders. We wish to explore the efficacy of active and collaborative learning approaches including POGIL-like in a focused CI environment using the Agile methodology.

8. References

Anfara Jr, V. A., Brown, K. M., & Mangione, T. L. (2002). Qualitative analysis on stage: Making the research process more public. In (Vol. 31, pp. 28–38). Sage Publications Sage CA: Thousand Oaks, CA.

Aniche, M., Hermans, F., & Deursen, A. (2019). Pragmatic software testing education. In *Proceedings of the 50th acm technical symposium on computer science education (sigcse '19), acm* (p. 414–420). New York, NY, USA.

Bijlsma, L., Passier, H., Pootjes, H., Stuurman, S., & Doorn, N. (2020). *How do students test software units? part one: Their natural attitude diagnosed* (Technical Report. Open Universiteit,). Faculty of Science, Department of Computer Science.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. In (Vol. 13, pp. 4–16). Sage Publications Sage CA: Thousand Oaks, CA.

Buffardi, K., & Edwards, S. (2014). A formative study of influences on student testing behaviors. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 597–602).

Chen, Z., Zhang, J., & Luo, B. (2011). Teaching software testing methods based on diversity principles. In *2011 24th ieee-cs conference on software engineering education and training (csee t* (p. 391–395). Honolulu, HI, USA.

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

Clarke, P., Pava, J., Davis, D., Hernandez, F., & King, T. (2012). Using wrestt in se courses: An empirical study. In *Proceedings of the 43rd acm technical symposium on computer science education (sigcse '12), acm* (p. 307–312). New York, NY, USA.

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.

Drake, J., & Drake, J. (2003). Teaching software testing: Lessons learned. Citeseer.

Edwards, S. (2004). Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th sigcse technical symposium on computer science education* (p. 26–30).

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).

Elbaum, S., Person, S., Dokulil, J., & Jorde, M. (2007). Bug hunt: Making early software testing lessons engaging and affordable. In *29th international conference on software engineering (icse'07)* (pp. 688–697).

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

Florea, R., & Stray, V. (2020). A qualitative study of the background, skill acquisition, and learning preferences of software testers. In *Proceedings of the evaluation and assessment in software engineering* (pp. 299–305).

Foley, D. E. (2002). Critical ethnography: The reflexive turn. In (Vol. 15, pp. 469–490). Taylor & Francis.

Garousi, V., & Zhi, J. (2013). A survey of software testing practices in canada. In (Vol. 86, p. 1354–1376).

Gopal, B., & Cooper, S. (2022). POGIL-like Learning in Undergraduate Software Testing and DevOps - A Pilot Study. In *Proceedings of the 27th annual acm conference on innovation and technology in computer science education (iticse)* (p. Accepted.).

Gopal, B., Cooper, S., Olmanson, J., & Bockmon, R. (2021). Student difficulties in unit testing, integration testing and continuous integration: An exploratory pilot qualitative study..

Greising, L., Bartel, A., & Hagel, G. (2018). Introducing a deployment pipeline for continuous delivery in a software architecture course. In *Proceedings of the 3rd european conference of software engineering education* (p. 102–107).

Hanson, D. (2005). Designing process-oriented guided-inquiry activities. In (pp. 1–6).

Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative health research*, *15*(9), 1277–1288.

Hsu, L. M., & Field, R. (2003). Interrater agreement measures: Comments on kappan, cohen's kappa, scott's π, and aickin's α. In (Vol. 2, pp. 205–219). Taylor & Francis.

Hu, H., Kussmaul, C., Knaeble, B., Mayfield, C., & Yadav, A. (2016). Results from a survey of faculty adoption of process oriented guided inquiry learning (pogil) in computer science. In *Proceedings of the 2016 acm conference on innovation and technology in computer science education* (pp. 186–191).

Hu, H., & Shepherd, T. (2013). Using pogil to help students learn to program. In (Vol. 13, pp. 1–23). ACM New York, NY, USA.

Hu, H., & Shepherd, T. (2014). Teaching cs 1 with pogil activities and roles. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 127–132).

Kennedy, C., & Kraemer, E. T. (2019). Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 acm conference on innovation and technology in computer science education* (pp. 224–230).

Ketelhut, D. J., & Schifter, C. C. (2011). Teachers and game-based learning: Improving understanding of how to increase efficacy of adoption. In (Vol. 56, pp. 539–546). Elsevier.

Kussmaul, C. (2011). Process oriented guided inquiry learning for soft computing. In *International conference on advances in computing and communications* (pp. 533–542).

Kussmaul, C. (2012). Process oriented guided inquiry learning (pogil) for computer science. In *Proceedings of the 43rd acm technical symposium on computer science education* (pp. 373–378).

Lemos, O., Ferrari, F., Silveira, F., & Garcia, A. (2015). Experience report: Can software testing education lead to more reliable code?. In *2015 ieee 26th international symposium on software reliability engineering (issre)* (pp. 359–369).

Lincoln, Y., & Guba, E. (2006). *Naturalistic inquiry*. Newbury Park: Sage Publications.

Maher, M. L., Latulipe, C., Lipford, H., & Rorrer, A. (2015). Flipped classroom strategies for cs education. In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 218–223).

Mazur, E. (1997). *Peer instruction a user's manual*. Prentice Hall.

Memar, N., Krishna, A., McMeekin, D. A., & Tan, T. (2018). Gamifying information system testing–qualitative validation through focus group discussion.

Ng, S., Murnane, T., Reed, K., Grant, D., & Chen, T. (2004). A preliminary survey on software testing practices in australia. , 116–125.

Robson, C. (2002). *Real world research: A resource for social scientists and practitioner-researchers*. Wiley-Blackwell.

Stray, V., Florea, R., & Paruch, L. (2021). Exploring human factors of the agile software tester. *Software Quality Journal*, 1–27.

Taipale, O., & Smolander, K. (2006). Improving software testing by observing practice. In *Proceedings of the 2006 acm/ieee international symposium on empirical software engineering* (pp. 262–271).

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).

Yadav, A., Kussmaul, C., Mayfield, C., & Hu, H. (2019). Pogil in computer science: Faculty motivation and challenges. In *Proceedings of the 50th acm technical symposium on computer science education* (pp. 280–285).