How Effective and Efficient Are Student-Written Software Tests?

Amanda Showler
Ontario Tech University
Oshawa, Canada
amanda.showler@ontariotechu.net

Michael A. Miljanovic
Ontario Tech University
Oshawa, Canada
michael.miljanovic@ontariotechu.ca

Jeremy S. Bradbury
Ontario Tech University
Oshawa, Canada
jeremy.bradbury@ontariotechu.ca

ABSTRACT

Many computer science students complete their undergraduate degrees with insufficient testing skills and knowledge. To understand the gaps in students' testing skills and knowledge, we analyzed 1014 software tests written by 12 groups in an undergraduate Software Quality Assurance (SQA) course project. In the project the student groups were provided a requirements document and were instructed to follow Test Driven Development (TDD) practices using black-box tests. To understand how the groups applied black-box testing in their project, we created an automatic tool to sort the tests into categories or "test buckets." By analyzing the test bucket data, we were able to assess the effectiveness and efficiency of student-written tests. We observed that the student groups were significantly more likely to test for explicit requirements than implicit requirements and significantly more likely to test happy paths than invalid inputs. Furthermore, students inefficiently tested happy paths, invalid inputs and explicit requirements resulting in a higher proportion of software tests with duplicate intent. Based on these results we provide insights into how black-box test education can be improved.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Applied computing → Education**; • **Social and professional topics → Software engineering education**.

KEYWORDS

black-box testing, software testing, testing education

ACM Reference Format:
Amanda Showler, Michael A. Miljanovic, and Jeremy S. Bradbury. 2025. How Effective and Efficient Are Student-Written Software Tests?. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701809

1 INTRODUCTION

The issue of insufficient software knowledge in recent computer science graduates has been documented in the literature for more than 20 years [20, 21]. The need for industry-relevant software education has also been identified [15]. This knowledge gap is especially concerning for software testing education when one considers the relationship between software testing and the development of high quality software.

Software testing is a challenging topic to learn and a challenging topic to teach [3]. To improve the quality of software testing education, researchers have focused on different strategies including adopting test driven development early in computer science curriculum [12], utilizing testing checklists [4], running student tests against other student's code [18], increasing student engagement through gamification [16] and providing high quality feedback in software test education tools [11, 19]. In addition to new approaches to software testing education, there has also been empirical research focused on better understanding testing education including assessing software testing feedback from instructors and teaching assistants [3], surveying perceptions of students and teaching assistants [3], studying software testing mistakes by students versus industry [14] and analyzing student test suites [13].

Most of the software testing education research has focused on analyzing secondary data such as instructor feedback or has focused on surveys about test education perceptions. Very little research has studied actual software tests written by students to understand the gaps in testing education. This observation is supported by a recent survey in which Garousi et al. found very little work on the analysis of tests to understand student mistakes [17]. When research does focus on student-written tests, it focuses almost exclusively on unit testing [1, 5, 9, 10, 13]. Furthermore, there is a lack of data sets of student-written tests to analyze. This lack of data sets and analysis of student written software testing artifacts is contrasted by the rich literature on data sets and analysis of student written computer programs. For example, long running projects like Blackbox have collected several terabytes of source code data that are available to analyze [7, 8].

To address this issue, we aim to assess both the effectiveness and efficiency of black-box tests written by students in a senior undergraduate elective course on Software Quality Assurance (SQA). In particular, we are interested in students ability to test explicit versus implicit requirements and happy path versus invalid input tests. We advocate that identifying the types of test students struggle with can provide valuable insight to educators. Furthermore, by adjusting teaching materials and learning activities to address the gaps observed in student tests, educators can improve software testing education practices.

2 A SOFTWARE QUALITY ASSURANCE COURSE PROJECT

The data discussed and analyzed in our research was produced in a senior undergraduate elective course on software quality assurance (SQA). The course spends approximately 50% of the lectures on the topic of software testing and includes a major course project worth

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0531-1/25/02…$15.00
https://doi.org/10.1145/3641554.3701809

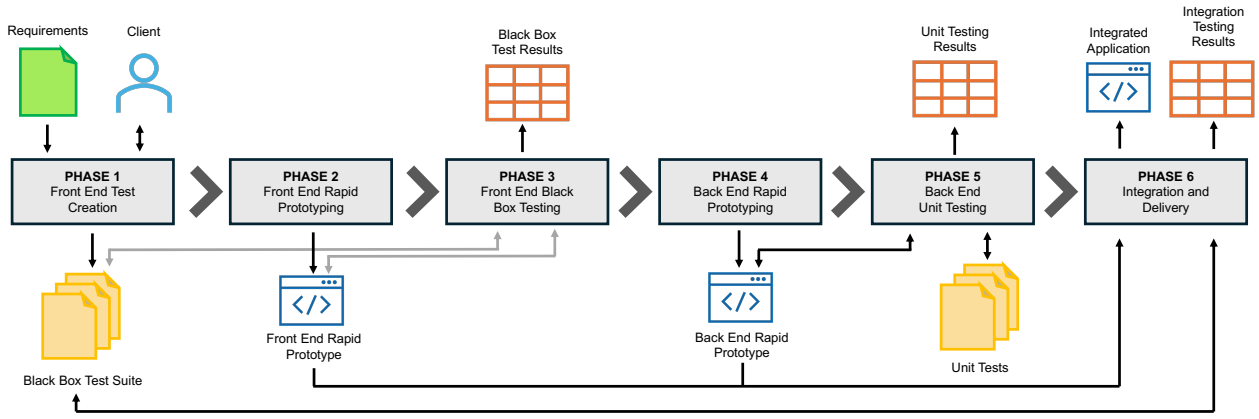


Figure 1: Software Quality Assurance Course Project Phases

60% of the student's final grade. The SQA course project is a full stack development project designed to emulate the experience and challenge of working in a real industry software team. Students work in groups of 3-4 students, and are instructed to complete the project using agile development practices based on Extreme Programming [6]. Most of the assessment is not based on code correctness, but instead on adherence to agile practices, the quality of source code and the completeness of the testing. The project is based on the development of a software system with a defined front end and back end that is specified in a requirements document. The system is transactional in nature and varies from year-to-year. For example, the most recent system was an auctioning system that allowed users and administrators to input front-end transactions via text or file input, and merging and storing all data in local file storage via the back end. Other past projects include the development of a banking application and a ticketing system. The intent of the project design is not to focus on the development of a novel software system, but is instead to enforce best software design practices and encourage the use of a consistent software development process.

The requirements document for the project is intended to be a full software specification and includes both requirements and constraints for the front-end transactions as well as any required file formats. Not all requirements are explicit – some are implicit. For example, in the auctioning system, there is no explicit requirement that a bid cannot be negative or zero. Furthermore, the requirements document includes omissions and inconsistencies (both planned and unplanned). To address issues with the requirements, the groups have access to a "client" account via a course Slack[1] channel. One of the key learning outcomes of the project is for students to improve their professional communication skills by asking questions as needed. Groups must address the client's feedback and clarifications in their development.

To support the development process, groups are required to utilize GitHub for version control. In addition to committing all source code, tests, scripts, and documentation to GitHub, there are six biweekly phases to the project which each have their own defined deliverables (see Fig. 1):

- **Phase 1:** Front-end black-box test creation for test-driven development. The black-box test suites should provide requirement coverage based on both the specification document and the implicit requirements provided by the client. Students are evaluated on the completeness of their test suite, as well as adherence to input and output format specifications that are designed to ensure that all student projects are interchangeable. This is critical for later phases of the project, where students are required to integrate other students' code with their own.
- **Phase 2:** Rapid prototyping of the front-end component including a design specified using a UML class diagram and an implementation in C++ or Java.
- **Phase 3:** Automated black-box testing of the front-end prototype from Phase 2 using the test suite from Phase 1. This phase requires students to write automated test scripts, record test case results documenting all failure tests, causes of failures and fixes. During the testing process the test suite and the prototype are modified to address failed tests. The black-box testing is rerun until all tests pass. The iterative approach used in Phase 3, allows students to discover that either their code has errors, or their tests have errors (or both), and they are required to both fix the issues and report their results as part of their submission.
- **Phase 4:** Rapid prototyping of the back-end component. This phase has the same deliverables as Phase 2 except for the back end instead of the front end.
- **Phase 5:** Development of white-box unit tests for the back end. Results for the unit tests are recorded (similar to Phase 3) and the tests and/or prototype are fixed to address failed tests. This process is repeated until all tests pass.
- **Phase 6:** Integration and delivery involves merging the tested front end from another group (Phase 3 output) with the tested back-end component from Phase 5 to create a complete software system. The integration of another groups' front end with their own back end ensures that students experience working with and modifying other developers' code – a common activity in industry projects. The final

[1]https://slack.com/

integrated application is automatically tested using student-written test scripts. The integration tests are a combination of the black-box tests written in Phase 1 combined with other operational tests written specifically for integration testing. All test results are recorded, fixes made for failed tests with the process repeated until all tests pass. Once all integration tests are passed, the full software system is ready to be deployed.

3 METHODOLOGY

3.1 Data Collection and Analysis

While there are multiple types of test data produced in the above course project, the analysis in our research focuses on the black-box tests that have been successfully used to test the front-end component in Phase 3. We selected the test cases after Phase 3 because this increased the likelihood that each group's tests can be run on their completed front end. Recall that the test data was stored in each group's GitHub project repository.

To start, each group's test data was cleaned by standardizing the student's file organization, anonymizing the files, and creating a configuration file to document any minor implementation variations.

To analyze the data, we built an automated system that extracted a given group's test data and categorized the individual tests into test buckets. The test buckets are defined as groups of related test cases that are related to a specific requirement. All the test buckets have been predefined based on the requirements specification document and labelled as explicit requirements or implicit requirements.

In addition to mapping the test cases to requirement-based test buckets, the automated system also analyzes the test cases sequential to determine if input data is valid (happy path) or invalid based on the requirements document. If an invalid input is encountered, the test is labeled as an invalid test. If no invalid inputs are detected, a test case is labeled as a happy path test.

3.2 Data Exclusion Criteria

While best efforts were made to analyze all student test data, we ultimately did establish data exclusion criteria. First, if a group's test suite had a major deviation from the requirements, the test suite was excluded. A test suite with a major deviation was defined as being not suitable for testing another group's software system and would require considerable manual analysis to pre-process and analyze. Second, we decided to exclude outlier test buckets that were tested by at most one group.

3.3 Metrics

3.3.1 Test Suite Effectiveness. Test suite effectiveness is defined by coverage of the test buckets. To measure the effectiveness, the following equation is used:

- Let x be the distinct buckets covered by a given test suite.
- Let n be the number of total buckets possible based on the requirements.

$$\text{Effectiveness} = \overline{\left(\frac{x}{n}\right)} \times 100\%$$

3.3.2 Test Suite Efficiency. Test suite efficiency is defined by the proportion of the test suite that are distinct tests. To measure efficiency, the following equation is used:

- Let x be the distinct buckets covered by a given test suite.
- Let t be the total number of tests in the test suite.

$$\text{Efficiency} = \overline{\left(\frac{x}{t}\right)} \times 100\%$$

3.3.3 Metric Threshold. The use of metrics can be enhanced by defining threshold values. We define the thresholds values for the effectiveness and efficiency of test suites using the thresholds proposed by Alves et al.[2]. They define four threshold categories which are widely used for software testing metrics – *low* (0-70%), *moderate* (70-80%), *high* (80-90%), and *very high* (90-100%).

4 RESULTS

Test suites were collected from 17 groups in the Winter 2023 cohort of an upper-year elective software quality assurance course at Ontario Tech University. The 17 test suites were collected from GitHub repositories and after applying the exclusion criteria (see Section 3.2) we had 12 test suites available for analysis. These test suites included 1014 individual software tests and 49 requirements-based test buckets.

We will first consider the effectiveness and efficiency of the test suites based on the coverage of explicit vs. implicit requirements. Next we will analyze the effectiveness and efficiency with respect to covering explicit happy paths and invalid inputs. After presenting the effectiveness and efficiency results, we will discuss our results in the broader context of software testing education.

4.1 Explicit vs. Implicit Requirement Tests

In an effort to understand the effectiveness and efficiency of student test suites, we first analyzed the test suites based on the coverage of explicit and implicit requirement test buckets. We first analyzed the data using an unpaired t-test and found that the number of groups testing explicit requirement buckets (M = 9.20, SD = 1.15) was higher than those testing implicit requirement buckets (M = 4.45, SD = 2.19), with a significant difference observed, $t(38) = 8.59$, $p < 0.0001$. Next, we performed a different analysis using a paired t-test and found that explicit requirements testing is significantly more effective (M = 77.30, SD = 18.21) than testing for implicit requirements (M = 37.08, SD = 25.27), $t(11) = 4.78$, $p = 0.0006$. The results of which groups covered the specific explicit and implicit requirement buckets is presented in Figure 2. Explicit requirements were all tested by more groups than any of the implicit requirements (with one exception). This exception was the implicit requirement that the login should fail if the wrong username is given and was tested by all 12 groups Overall, the results highlight that students had difficulty identifying and testing implicit requirements.

After considering the coverage of the requirements, we next investigated the number of tests in each distinct requirements bucket. In other words, we looked at the amount of duplication in testing each explicit and implicit requirement. When we consider the issue of duplicate tests and over-testing, implicit requirements are more efficiently tested with fewer duplicates (see Figs. 4a & 4b.We analyzed the data using a paired t-test and found that the testing

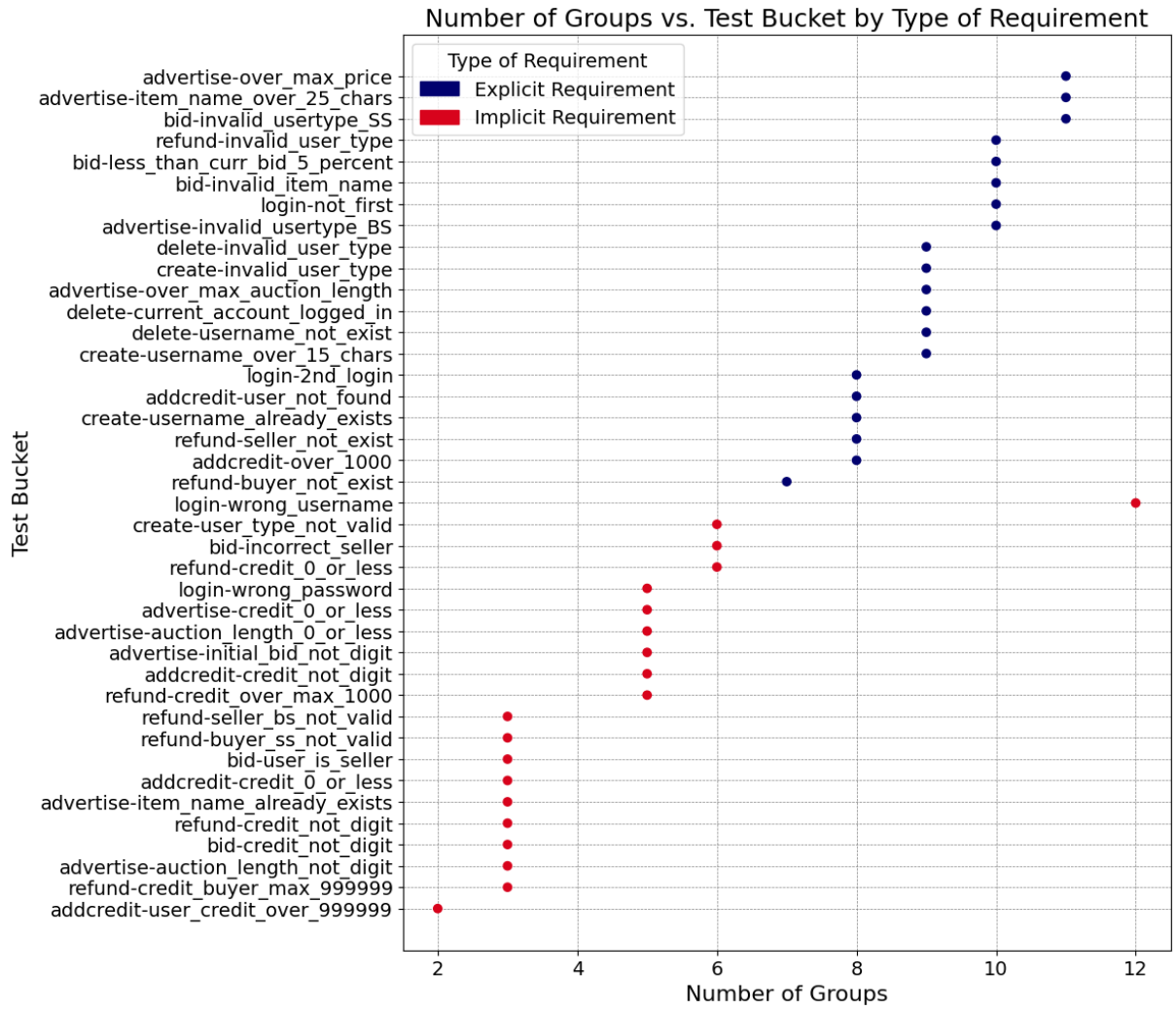


Figure 2: Number of Groups vs. Test Bucket by Type of Requirement

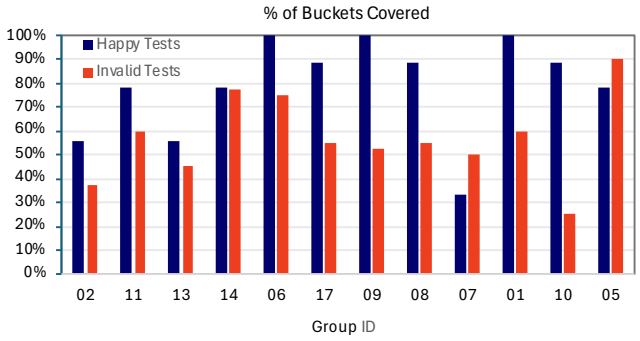
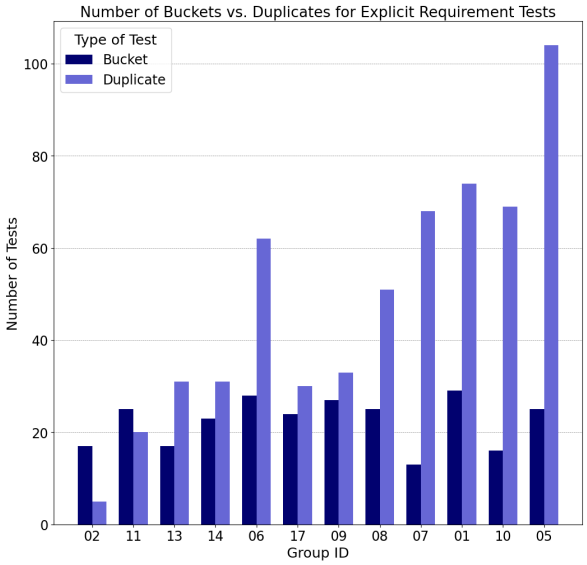


Figure 3: Percentage of Happy Test vs. Invalid Input Test Buckets Covered

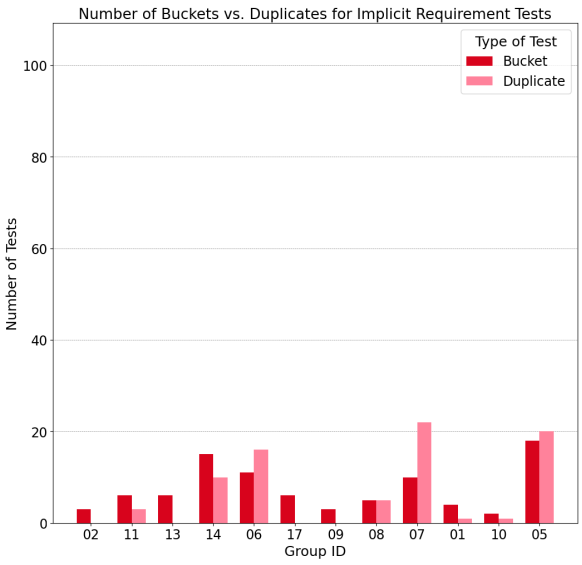
efficiency for implicit requirements (M = 70.22, SD = 25.39) was significantly higher than explicit requirements (M = 37.22, SD = 17.39), t(11) = 5.63, p = 0.0002. This indicates that the students' test suites have more duplicate tests for explicit requirements than implicit.

4.2 Happy Path vs. Invalid Input Tests

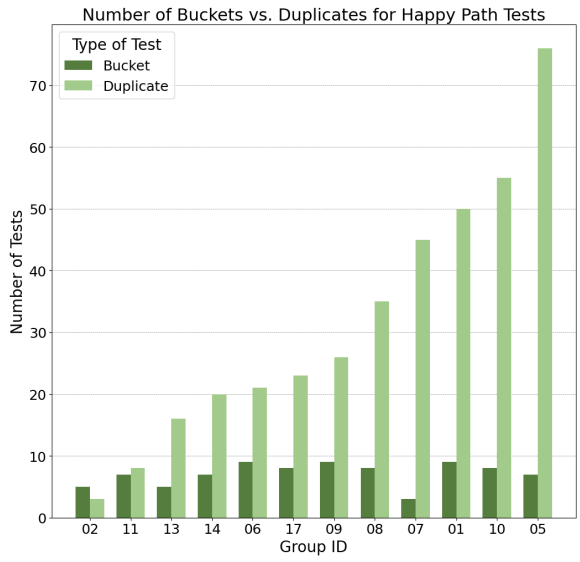
We also analyzed happy path test and invalid input tests with respect to effectiveness and efficiency. Overall, when we performed our analysis, we found that the 9 happy path test buckets were more effectively tested than the 40 invalid input test buckets (see Fig. 3). We used a paired t-test and found that the groups tested happy paths (M = 78.70, SD = 20.89) with a higher effectiveness than they tested invalid inputs (M = 56.88, SD = 17.78), with a significant difference observed, t(11) = 3.15, p = 0.0092. Therefore, the invalid input test buckets are more often missed by students than the happy path buckets.



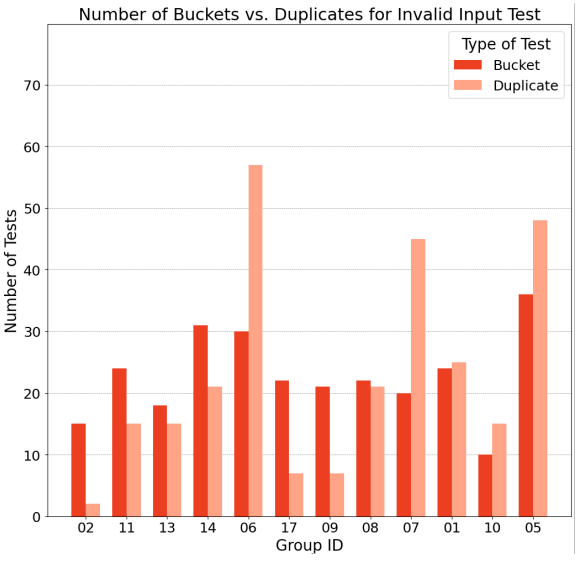
(a) Explicit Requirements Tests



(b) Implicit Requirement Tests



(c) Happy Path Tests



(d) Invalid Input Tests

Figure 4: Bucket Coverage vs. Duplicates by Group

Furthermore, the efficiency for testing invalid input buckets was significantly higher (M = 55.25, SD = 17.67) than the happy path buckets (M = 25.14, SD = 16.00), $t(11) = 8.24$, $p < 0.0001$ (see Figs. 4c & 4d). This supports the result that invalid input buckets have fewer redundant tests than happy path buckets.

4.3 Discussion

The results of the student-written tests in our dataset show some interesting trends that require further discussion, as well as confirm some of our expectations about where students might face challenges in testing (see Fig. 5).

When we consider the types of requirement that students test adequately, it is perhaps not surprising that implicit requirements are statistically not as well tested as explicit requirements. The identification and testing of implicit requirements are more likely to benefit from experience as well as domain knowledge of the software-under-test. In our case, we deliberately chose a system that would have similarity with existing software the students may have used. For example, an auction site would be very similar to a site like eBay. An important point to note is that despite requirements being implicit, there were groups that correctly identified the need to test these requirements.

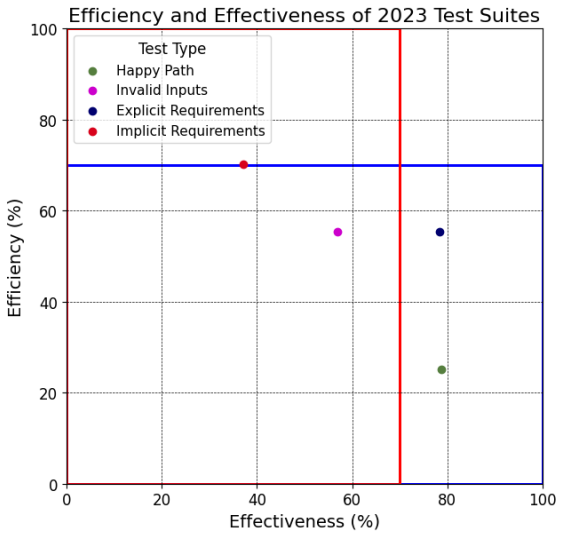


Figure 5: Effectiveness vs. Efficiency of Test Buckets

To understand why some student groups tested these requirements and some did not, it would be helpful to understand if the ineffective testing of implicit requirements is a result of a lack of domain knowledge or a lack of understanding of the need to find and test these requirements. Upon review of our course lecture materials, we devoted minimal time to the identification of requirements that need to be tested and instead devoted much of our attention to the different coverage metrics and strategies for creating tests. It is possible that there needs to be more focus and practice on the analysis of requirements to support the creation of test cases.

The test efficiency differences between explicit and implicit requirements are possibly related to the fact that students were more confident writing tests for requirements that were presented explicitly. There may also be a bias toward testing explicit requirements because they are viewed as more important since they have been documented in the requirements specification. The tendency to over-test explicit requirements may also demonstrate a need for more emphasis in the course on testing suite performance, as well as the practice of test maintenance where duplicate tests would be removed. In other words, more time in the classroom discussing why more tests are not always better.

When we consider happy path and invalid input tests, it is interesting to observe that happy path testing is both more effective and efficient. This might be due to the unbalanced frequency of invalid input (40) and happy path buckets (9). In other words, the students may have felt more of a need to balance out the overall number of positive and negative tests. Another consideration is that students may have first been interested in making sure that they created tests to demonstrate that the software worked correctly (happy path tests) and were less focused on invalid edge cases and negative cases that they viewed as less likely to occur (invalid input tests).

Our results show some of the trends around the effectiveness and efficiency of black-box student test creation. Unfortunately, we can only hypothesize at this point the reasons for these trends. A follow-up survey on students perceptions might be effective to understand what motivates these testing behaviours. Regardless of why students test software the way they do, the data we analyzed does provide educators with insights into potential gaps in students' testing knowledge and the application of that knowledge. Using this information to enhance lectures and practice problems is one way to address the ineffective and inefficient testing practices observed.

4.4 Threats to Validity

The main threat to validity we observed in our research is with respect to external validity. Our analysis is based on data from a specific project that occurs in one specific course, which may not be representative of student testing in other courses at other universities. Furthermore, the test suite evaluation was limited to black-box requirements coverage and does not generalize to other testing practices including unit testing and white-box testing.

5 CONCLUSION

Software testing is essential to the development of high quality software, and there is a lack of publicly available data and analysis of student-written tests. We believe that an evidence-based approach to software test education necessitates more research into students software testing knowledge, skills, and practices. In this paper, we evaluated student-written test suites from an undergraduate Software Quality Assurance (SQA) course project. Our evaluation focused on understanding the effectiveness and efficiency of black-box tests with an aim of using the data to provide insights into the enhancement of software testing education. In our evaluation, we observed that the student groups were significantly more likely to test for explicit requirements than implicit requirements and were also significantly more likely to test happy paths than invalid inputs. Student test data also showed that they inefficiently tested happy paths, invalid inputs and explicit requirements resulting in a higher proportion of software tests with duplicate intent. Overall, our results suggest that students have difficulty writing implicit requirement tests and, to a lesser extent, invalid input tests, as well as having difficulty reducing test suites. Based on the data, we suggest that the students in the study could benefit from a better understanding of how to create tests from a requirements document, more knowledge on the need to test both positive and negative inputs, and an understanding of the importance of software test performance.

In the future, we plan to gather more data from future instances of the SQA course and obtain student permission to use their anonymized test data to create a public repository that can be used by other researchers. We would also like to expand the study to consider other kinds of tests (e.g., unit tests). We would also like to explore the benefits of the automated system used to categorize the test data in our research as a feedback tool for students. For example, we could deploy the tool to provide automated feedback to students prior to project submission, giving them insight into the quality of their test suites.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), grant 2018-06588.

REFERENCES

[1] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. 2023. Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 29–39. https://doi.org/10.1109/ICSE-SEET58685.2023.00009

[2] T L Alves, C Ypma, and J Visser. 2010. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[3] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 414–420. https://doi.org/10.1145/3287324.3287461

[4] Gina R. Bai, Zuoxuan Jiang, Thomas W. Price, and Kathryn T. Stolee. 2024. Evaluating the Effectiveness of a Testing Checklist Intervention in CS2: An Quasi-experimental Replication Study. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3632620.3671102

[5] Gina R. Bai, Justin Smith, and Kathryn T. Stolee. 2021. How Students Unit Test: Perceptions, Practices, and Pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 248–254. https://doi.org/10.1145/3430665.3456368

[6] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional.

[7] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/2632320.2632343

[8] Neil C. C. Brown and Michael Kölling. 2020. Blackbox Mini - Getting Started With Blackbox Data Analysis. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1387. https://doi.org/10.1145/3328778.3367006

[9] Kevin Buffardi and Juan Aguirre-Ayala. 2021. Unit Test Smells and Accuracy of Software Engineering Student Test Suites. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 234–240. https://doi.org/10.1145/3430665.3456328

[10] Kevin Buffardi, Pedro Valdivia, and Destiny Rogers. 2019. Measuring Unit Test Accuracy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 578–584. https://doi.org/10.1145/3287324.3287351

[11] Lucas Cordova, Jeffrey Carver, Noah Gershmel, and Gursimran Walia. 2021. A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 87–93. https://doi.org/10.1145/3408877.3432417

[12] Chetan Desai, David S. Janzen, and John Clements. 2009. Implications of integrating test-driven development into CS1/CS2 curricula. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 148–152. https://doi.org/10.1145/1508865.1508921

[13] Stephen H. Edwards and Zalia Shams. 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* (Uppsala, Sweden) *(ITiCSE '14)*. Association for Computing Machinery, New York, NY, USA, 171–176. https://doi.org/10.1145/2591708.2591757

[14] Sigrid Eldh, Hans Hansson, and Sasikumar Punnekkat. 2011. Analysis of Mistakes as a Method to Improve Test Case Design. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 70–79. https://doi.org/10.1109/ICST.2011.52

[15] Sigrid Eldh and Sasikumar Punnekkat. 2012. Synergizing industrial needs and academic research for better software education. In *2012 First International Workshop on Software Engineering Education Based on Real-World Experiences (EduRex)*. 33–36. https://doi.org/10.1109/EduRex.2012.6225703

[16] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019. Gamifying a Software Testing Course with Code Defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 571–577. https://doi.org/10.1145/3287324.3287471

[17] Vahid Garousi, Austen Rainer, Per LauvÃ¥s, and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *Journal of Systems and Software* 165 (2020), 110570. https://doi.org/10.1016/j.jss.2020.110570

[18] Michael H. Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 271–275. https://doi.org/10.1145/563340.563446

[19] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software* 175 (2021), 110905. https://doi.org/10.1016/j.jss.2021.110905

[20] Silvana M. Melo, Veronica X. S. Moreira, Leo Natan Paschoal, and Simone R. S. Souza. 2020. Testing Education: A Survey on a Global Scale. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 554–563. https://doi.org/10.1145/3422392.3422483

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