Integrating Testing Throughout the CS Curriculum

Sarah Heckman
North Carolina State University
sarah_heckman@ncsu.edu

Jessica Young Schmidt
North Carolina State University
jessica_schmidt@ncsu.edu

Jason King
North Carolina State University
jtking@ncsu.edu

***Abstract*—Software testing is a critical component of any software development lifecycle, but becoming an experienced software tester requires understanding many strategies for writing high-quality test cases and a significant amount of practice. *Situated learning theory* suggests that students should be exposed to things they would see in a professional workplace. In terms of software testing, students should be exposed to real-world software testing practices in a variety of contexts, from the simplest of programs to the very complex. *The goal of this paper is to share our experience integrating software testing into our undergraduate curriculum at North Carolina State University.* In this paper, we discuss how software testing is taught in our CS1 – Introductory Programming, CS2 – Software Development Fundamentals, and several other courses beyond CS2. Over the past 10 years of teaching software testing in introductory programming courses, we discuss lessons learned and highlight open concerns for future research.**

***Index Terms*—testing, CS1, CS2**

I. Introduction

In computer science education, software testing is commonly used to check that a piece of student-engineered software meets the instructor-provided requirements specification and functions correctly. Instead of assuming the role of software tester *for* student-engineered software, instructors should encourage students, themselves, to actively engage in the software testing process to help with debugging, validation, and verification efforts [1]. Since any piece of software can be tested, software testing should be taught from day 1 alongside `HelloWorld` to help students adopt a mindset of outlining expected results beforehand and comparing against observed actual outputs. However, students need strategies for testing to help them successfully determine how to write high-quality test cases and to identify when they are "done" working on a piece of software.

The goal of this paper is to share our experience integrating software testing into our undergraduate curriculum at North Carolina State University. In this paper, we discuss the following contributions:

- an overview of testing integrated throughout an undergraduate computer science curriculum;
- an open-source testing packet that instructors can integrate into introductory programming courses;
- lessons learned in integrating testing in introductory courses; and
- open problems with integrating testing in computer science classrooms.

II. Teaching and Evaluating Testing

We have integrated software testing instruction in several of the introductory core courses that students take for the Computer Science major or minor at North Carolina State University (NCSU). In our CS1 and CS2 courses, we provide students with instruction on black-box and white-box testing practices and explore testing further through automation, coverage, and performance analysis. Students continue to test throughout the computer science curriculum. The integration of testing into coursework at all levels utilizes *situated learning theory* [2] where students are exposed to professional practice [3]. Our testing instruction includes written system test plans that complement the implementation and execution of test code, frameworks, and harnesses. As students gain maturity in the computer science discipline, we expect their communication skills about software testing to progress [4].

Since most introductory programming textbooks do not formally cover testing, we have created our own supplemental testing reading and lecture materials.[1] Creating a common set of materials for our introductory course is important because not all instructors have a background in software engineering – we provide a standard set of instructional materials to ensure that students have a common background in testing. Additionally, future courses continue to build upon the testing materials using common vocabulary as more advanced testing topics are presented throughout the curriculum.

A. CS1: Introductory Programming

Our CS1 course introduces students to the Java programming language through the objects-late paradigm of teaching. CS1 course is taught in a combined lecture/lab setting with 30-60 students per section. Students have access to a programming environment and practice course topics through small daily coding activities while receiving real-time face-to-face help from the teaching staff. Prior to formally introducing testing, students learn about data types, variables, `for` loops, parameters, return values, use of objects (e.g., `String` and `Scanner`), conditional statements, and text processing. While students are not writing formal tests for these lessons, they are encouraged to test the assigned programs with various inputs by describing expected results, executing programs multiple times, and observing the actual outputs of the software. Testing is formally introduced during two lecture periods about halfway through the semester. The first testing lecture covers

[1]Current materials are available at https://sos-cer.github.io/testing/.

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00079

the purpose of testing and techniques for black-box testing, including strategies for testing against requirements and writing test cases based on equivalence classes, boundary values, and diabolical testing. The second testing lecture includes discussion on white-box testing (using JUnit), visualizing control-flow for structuring test cases around paths through the code, and understanding the relationship between testing and debugging. During the two testing lectures, students practice black-box and white-box testing skills by completing activities based on a simplified version of Bruce Schneier's card-based Solitaire encryption algorithm [5].

Once software testing is formally introduced, students are required to submit a specific number of black-box and/or white-box tests for each of the remaining assignments in the course. Test cases must be non-redundant, repeatable, and specific. To simplify and automate grading white-box tests, students add each test to its own test method.

To help review course materials, students complete a comprehensive exercise during the final two weeks of class. The comprehensive exercise requires students to iterate through the full software development lifecycle, including testing [6]. Students often state that the exercise helped them better understand the importance of testing.

B. CS2: Software Development Fundamentals

Our CS2 course provides a deeper look at software engineering, especially testing. The CS2 course is taught in a large lecture hall environment with over 100 students per section. Students must also enroll in separate, smaller weekly lab sections with 25-30 students. Testing is covered in the second lecture of the course and is used in *all* assignments in the course. Instruction goes beyond black-box and white-box testing to introduce students to unit, integration, system, acceptance, and regression testing. The third lecture of the course introduces test coverage and static analysis. Students integrate testing into a software development lifecycle with supporting processes and tools as shown in Figure 1.

Students complete three Guided Projects and 12 Labs over the course of the semester[2]. Unit and system tests are provided for Guided Projects 1 & 2. Starting with Guided Project 3 and Lab 2, students are expected to write and run their own system and unit tests to evaluate how well their implementations meet the requirements and imposed designs. For projects, students write their system tests in a black box test plan document [4] (along with a design proposal) for a given set of requirements for Part 1. In Part 2, students implement and unit test a provided design. At the end of the project, they are expected to run their (possibly revised) system tests and report the actual results of execution. We assess the quality of students' tests using a combination of code coverage, static analysis tools to find problematic tests[3], manual test code inspection, and

[2]Guided Projects and Labs are available at: https://soscer.github.io/projects/labs.html

[3]We consider a test class with no test methods and a test method with no asserts to be problematic in our introductory courses when using JUnit.

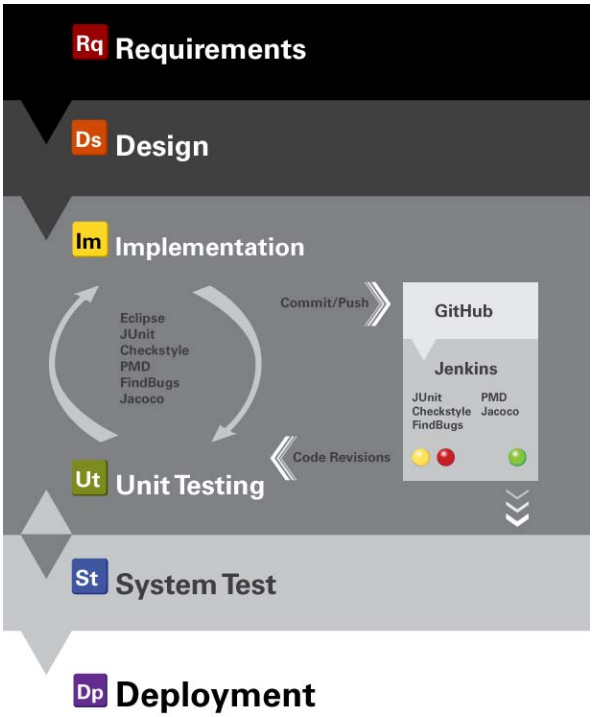

Fig. 1. Student Development Lifecycle and Tools

evaluation of system test plans. Students' solutions are also evaluated against the teaching staff unit and acceptance tests.

C. Testing Beyond CS1 and CS2

Students continue using their testing skills beyond our introductory courses. In Data Structures and Algorithms, students continue writing their own unit and system tests and consider the runtime performance of algorithms when creating tests. For example, using a continuous integration server, the instructor can create a common execution environment that benchmarks the runtimes of the instructor's software solution. Using these benchmarks, the continuous integration server can then execute student software and impose timeouts to help encourage students to further improve the runtime efficiency of their algorithms or to encourage students to use more efficient data structures. In our C and Software Tools course, students learn more about scaffolding automated testing through provided test harnesses. Additionally, students learn that testing strategies are independent of language-specific tools. In our software security course, students formally document security test plans to locate potential security vulnerabilities in software. Understanding software testing helps students adopt the mindset of "thinking like an attacker" since penetration testers and attackers often exploit systems by changing inputs into the software and observing outputs to help craft attacks. In our software engineering course, students are introduced to front-end web testing frameworks where they can automate

their system-level tests [7], specifically Selenium [8] and Cucumber [9]. Since students learned testing basics earlier in the curriculum, we can focus instruction on behavior-driven development and more complex testing frameworks. Finally, students are expected to choose appropriate testing frameworks, automation, and report out on their unit, integration, and system testing in our senior design capstone course as the culmination of test progression through the curriculum [4].

III. Automating Testing

Automating test execution and evaluation is important for scaling timely evaluation and feedback as computer science course enrollments grow. Continuous integration systems help facilitate automating test execution and serve as examples of professional tools that students may see in the workplace. There are a number of commercial (e.g., Gradescope [10] and My Programming Lab [11]) and academic (e.g., Web-CAT [12] and AutoGrader [13]) automated grading systems in use that execute instructor and/or student tests as part of evaluation. We utilize professional tools to support out automated testing system following *situated learning theory* [2]. Students submit their code and tests to an enterprise GitHub [14] and their code is evaluated using scripts on the continuous integration system Jenkins [15]. Test cases written by the teaching staff are often incorporated into the build process to help evaluate student software deliverables for correctness. We provide more details and information out our automated grading setup in [16].

A. CS1: Introductory Programming

For evaluation and grading in CS1, the teaching staff uses a Jenkins continuous integration system behind-the-scenes. Every time a student pushes their code to GitHub, Jenkins will pull the student's code and 1) compile the student's code and tests, 2) compile the teaching staff tests against the student's code, 3) run Checkstyle [17] to ensure student-written code meets required style guidelines, 4) run the student's test cases, and 5) run the teaching staff's test cases to check the student's software for correctness. Students are not aware of the existence of the continuous integration system, nor do they have access to the system to receive any feedback about their software builds. However, during the Fall 2019 semester, students started receiving feedback about their builds (for example, whether the program was named correctly, compiled without errors, or contained any style alerts) in a branch of their GitHub repository for the last project of the semester. Future work will investigate the impact of the feedback on student work.

B. CS2: Software Development Fundamentals

For CS2, students are introduced to continuous integration as a tool to facilitate automated grading and feedback. Every time a student pushes their code to GitHub, Jenkins will pull the student's code and 1) compile the student's code and tests, 2) compile the teaching staff tests against the student's code, 3) run static analysis tools, 4) run the student's tests instrumented for coverage, and 5) run the teaching staff tests. If any one of these steps fails, then the build stops.

We have several checks in place during the build to encourage students in writing high-quality tests. We utilize several custom PMD [18] notifications adopted from Web-CAT [12] that find test classes without any test methods and test methods without any `assert*()` statements. If we see any of these static analysis notifications, teaching staff test results are not shown to students. We additionally check for coverage. Students are expected to have 80% statement coverage on each non-user interface class. Teaching staff tests for a class are not revealed until students have reached sufficient statement coverage. One challenge that we have noticed is that students will write poor tests to increase coverage and then use the teaching staff tests to debug their programs. While this is a discouraging practice, students will typically write better tests when debugging their programs by using the teaching staff hints about the test scenario that are provided for a test failure.

C. Testing Beyond CS1 and CS2

We utilize our automation framework in Data Structures and Algorithms, C and Software Tools, and our Software Engineering classes. As students progress through the curriculum, the amount of teaching staff testing support declines; students are expected to automate their own tests. Many students use continuous integration systems to automate their test builds in our Senior Design capstone course.

IV. Lessons Learned

We have learned several lessons over the years about how best to integrate testing into various levels of the curriculum.

Adoption. We have codified testing instruction through course learning outcomes. Testing materials were created by and are supported by software engineering faculty.

Tooling. When we first integrated automated unit testing into CS1, we intentionally decided to not use JUnit to minimize the tools that students are using. We structured our test automation instruction and templates to be similar in structure to JUnit tests to ease the transition to JUnit in CS2. Several years ago we moved to using JUnit in CS1 to remove the artificial testing structure we were imposing. The switch to using JUnit in CS1 had the additional benefit of introducing students to working with libraries when compiling and executing code from the command line.

Common tools and language. Identifying a common set of tools (where appropriate) and vocabulary to use around software testing was important for adopting testing across the curriculum.

Testing examples for instruction. The examples used to drive testing instruction are critical for students to see the value of testing in practice. When we first created our testing materials we used a program scenario of identifying if a triangle is equilateral, scalene, or isosceles given three sides of a valid triangle. However, students could easily identify logic errors in the provided codebase without ever writing any test cases because the problem was very simple. Students 1) did

not fully explore the requirements to create tests and 2) rarely saw how test failures can help find bugs in implementation. We later moved to a simplified version of the Solitaire encryption algorithm [5], which is much more complex than the triangle classification scenario and helps students understand the relationship between testing and debugging.

Automation. Automation is critical to support learning through rapid feedback and efficient grading by teaching staff.

Gaming automated grading. When teaching staff tests are provided, students try to obtain access to the results of those test cases as quickly as possible. Even with minimum statement coverage and static analysis thresholds required, students write poor quality tests to meet the thresholds and expose teaching staff test results as quickly as possible. Students can implement a working solution from teaching staff test feedback and will sometimes supplement their tests with their own version of the teaching staff scenario, but the quality of student test cases remains an issue. Adding additional checks for poor testing practice [16] and holding back some tests may minimize gaming.

Test flakiness. As software requirements and testing frameworks become more complex, poor testing practices can lead to test flakiness [19]. Providing additional examples, best practices to avoid flaky tests, utilizing test inspections, and cleaning up poorly written tests is critical to success of student projects, especially the one we use in software engineering that is built on over multiple semesters [7].

V. Open Problems

While we feel that we have a strong integration of testing throughout the curriculum, we believe there are open problems in testing education that need to be addressed.

Enforcing high-quality test cases when using automated grading systems. When teaching staff tests are provided, students avoid writing high quality tests; instead they depend on the automated grading system [20]. Coverage as a metric to assess test quality is easily gamed through writing poor tests that execute lots of code under test (or by writing lots of junk code that tests execute). Manual inspection for test quality, especially at scale, is challenging and, without automation, cannot happen during development preventing rapid feedback to students. An open problem in software testing is the automated assessment of test quality.

Using all-pairs testing of student tests against other students' code can identify if a test suite has bug revealing capability [21]. Running all-pairs analysis during submissions, especially close to project deadlines may be too costly. Mutation testing is another strategy but does not perform as well as all-pairs and is also costly for in-process feedback [21].

Debugging. Students struggle with debugging and we have observed that part of the reason appears to be due to writing poor tests. By providing feedback on test quality during automated feedback and grading, students can write better tests that will provide a better foundation for debugging work.

VI. Conclusions and Future Work

Software testing should not be treated as an afterthought. Fixing errors in software at the end of the software development lifecycle requires more time and resources than if errors were found earlier. With early coverage of testing topics, students have more time to learn and practice software testing skills throughout the undergraduate curriculum and recognize testing as a *continuous* activity that is a part of writing *any* size piece of software. We will continue to refine our testing instruction at all levels including work to address the open problems around automatically assessing test quality and debugging skills.

References

[1] S. H. Edwards, "Rethinking computer science education from a test-first perspective," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 148–155.

[2] J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, 1991.

[3] A. Begel and B. Simon, "Struggles of new college graduates in their first software development job," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 226–230. [Online]. Available: http://doi.acm.org/10.1145/1352135.1352218

[4] M. Carter, R. Fornaro, S. Heckman, and M. Heil, "Creating a progression of writing, speaking, teaming learning outcomes in undergraduate computer science/software engineering curricula," in *World Engineering Education Forum (WEEF)*, 2012.

[5] B. Schneier, "The solitaire encryption algorithm," May 1999. [Online]. Available: https://www.schneier.com/academic/solitaire/

[6] J. Y. Schmidt, "Reviewing cs1 materials through a collaborative software engineering exercise: An experience report," in *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*, 2020, p. to appear.

[7] S. Heckman, K. Stolee, and C. Parnin, "10+ years of teaching software engineering with itrust: the good, the bad, and the ugly," in *ICSE-SEET 2018*, 2018, pp. 1–4.

[8] "Selenium: Browser automation," http://www.seleniumhq.org/, 2017.

[9] "Cucumber," https://cucumber.io/, 2019.

[10] "Gradescope," https://gradescope.com/, 2017.

[11] "My programming lab," http://www.pearsonmylabandmastering.com/northamerica/myprogramminglab/, 2017.

[12] "Web-cat," http://web-cat.org/, 2017.

[13] M. T. Helmick, "Interface-based programming assignments and automatic grading of java programs," in *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*.

[14] "Github: How people build software," http://github.com/, 2017.

[15] "Jenkins: Build great things at any scale," https://jenkins.io/, 2017.

[16] S. Heckman and J. King, "Developing software engineering skills using real tools for automated grading," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: ACM, 2018, pp. 794–799.

[17] "Checkstyle," http://checkstyle.sourceforge.net/, 2017.

[18] "Pmd," https://pmd.github.io/, 2017.

[19] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "Wait wait. no, tell me: Analyzing selenium configuration effects on test flakiness," in *Proceedings of the 14th International Workshop on Automation of Software Test (AST '19)*, 2019, pp. 7–13.

[20] K. Buffardi and S. H. Edwards, "Reconsidering automated feedback: A test-driven approach," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*.

[21] S. H. Edwards and Z. Shams, "Comparing test quality measures for assessing student-written tests," in *Companion Proceedings of the 36th International Conference on Software Engineering*.