Automating Systems Course Unit and Integration Testing

Experience Report

Dee A. B. Weikle
James Madison University
Harrisonburg, VA
weikleda@jmu.edu

Michael O. Lam
James Madison University
Harrisonburg, VA
lam2mo@jmu.edu

Michael S. Kirkpatrick
James Madison University
Harrisonburg, VA
kirkpams@jmu.edu

ABSTRACT

Introducing software testing has taken on a greater importance in undergraduate computer science curricula in the last several years, with many departments using JUnit or other testing frameworks in the programming sequence and software engineering courses. We have developed an automated framework for unit and integration testing and grading for our intermediate-level systems course projects. Our system–designed to test C programs–combines the Check unit testing framework, custom Bash scripts for integration testing, and the Valgrind Memcheck memory leak detection tool. Although our courses use Linux, the framework is platform-independent and has been tested on a variety of other platforms.

We have used this framework for seven semesters with four different instructors as part of the computer science program at a primarily undergraduate university with an emphasis on liberal arts. We distribute both public and private tests so that students get immediate feedback on their progress without knowing the actual contents of every test. We have observed that knowing their code is not completely working motivates more students to figure out what they don't understand before the project deadline. It also gives students examples of different levels of tests to use to debug their code, encourages them to develop a deeper understanding of the project specification, and reduces student anxiety about grades.

ACM Reference Format:
Dee A. B. Weikle, Michael O. Lam, and Michael S. Kirkpatrick. 2019. Automating Systems Course Unit and Integration Testing: Experience Report. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287324.3287502

1 INTRODUCTION

Software testing has become a fundamental skill for computer science graduates entering the industry. Many employers expect their new hires to be familiar with methodologies that integrate testing and software development, such as test-driven development. Given this increased emphasis from industry, many computer science programs have explored ways to reinforce these concepts in courses beyond software engineering.

One common approach to increase students' exposure to testing practices is to bring unit testing into other domains, such as courses on human-computer interaction [3]. JUnit provides a standard industry framework for courses that use Java [12]; additionally, the rich features of JUnit facilitate using it for integration testing. In other areas, instructors have created their own languages and tools to bring unit testing to courses with C++ [2] or assembly language [7] projects. To support our intermediate-level systems courses, we have built a similar tool to facilitate automated testing in C.

Our system is designed to provide support for more than just unit testing. Specifically, our system provides an integrated framework for several automated testing components, including unit testing, integration testing, and dynamic analysis for memory leaks. Our system relies on common freely available tools, including the Check unit testing framework, the `diff` utility, and Valgrind's Memcheck memory leak detector. The framework uses Bash scripts to automate much of the processing. While our courses use Linux, the framework is platform-independent and has been tested on macOS and Cygwin for Windows.

Our design goals for this work go beyond just technical concerns, as well. In our experiences, many students struggle with systems courses for a variety of reasons. For most students, these courses are their first introductions to the C language and its related tools; consequently, they have no prior experience with explicit memory management. Most have minimal exposure to navigating the Linux command line. Furthermore, as our earlier courses are Java-based and assume no prior knowledge, many students are reluctant to move away from lightweight Java IDEs, such as Dr. Java or jGRASP.

By the end of the first semester in the systems sequence, our goal is for students to be able to develop command-line systems programs in C. Given the students' background, achieving this goal requires a significant amount of instructor support, both in the form of content scaffolding and motivational guidance. To facilitate both of these goals, our system provides students with all tests that will be used for evaluating their work. This includes a combination of public tests (to provide initial guidance) and private tests (precompiled and stripped of debugging symbols). Keeping the source code for the latter private is important, as many of these tests include code that would leak details about how to implement certain features.

A key feature of our approach is the structure of the test cases and their relationship to student grades. We adopt a specifications grading approach [10], in which subcomponents have a given specification that is evaluated in a pass/fail manner. The project requirements indicate which components are required to earn at least a grade of C, which for at least a B, and which are required for an A.

The advantage of this transparency is that it empowers students to monitor their progress independently.

This specifications grading approach has both cognitive and affective benefits for students. From the cognitive perspective, the grading structure provides scaffolding for students to learn how to break the project down into steps. They cannot earn a C or higher if *any* of the C tests fail, even if they pass several A or B tests; consequently, the grading structure guides them from simpler to more complex tasks. From the affective perspective, the tests serve as proximal subgoals [1] that reinforce motivation. Passing one test or group of tests increases the student's expectancy that they can pass the next, which is only slightly more complicated. And as the C tests are intentionally easy to pass with only modest effort, students quickly get the feedback–as they are working–that they are in no danger of failing.

Through course evaluations and informal discussions, we have observed anecdotal feedback that suggests students view the framework as helpful. Students have indicated that seeing the immediate feedback of their current grade status provided reassurance that kept them focused on making gradual progress. They have also identified the projects as being helpful preparation for exams by creating a platform to apply theoretical concepts.

We have used this framework for seven consecutive semesters with four different instructors leading the classes. Our department maintains a primary focus on undergraduate education within the context of a Master's university with a liberal arts emphasis. The majority of our students enter as computer science majors with little or no prior exposure to programming. Since adopting this testing framework in our systems curriculum, we have observed qualitatively improved student submissions and reduced student anxiety in these courses. In this experience report, we will describe some of the guiding principles of our approach, outline the major components of the system, and discuss lessons learned and tips for instructors who may want to use the system in their own courses.

2 BACKGROUND AND MOTIVATION

The automated testing tool that we present here is part of a larger departmental effort to redesign our systems curriculum [6]. Specifically, our aim has been to adopt a learner-centered philosophy that supports and encourages student autonomy within the constraints of a discipline with a rapidly growing and evolving body of knowledge. As part of this curriculum redesign, we have created a two-semester intermediate-level sequence that provides a foundation of the core knowledge for advanced systems work. Beyond normal cognitive learning outcomes, this two-semester sequence also emphasizes metacognitive objectives, with the aim that students leave these courses with the ability to use tools that are appropriate to the systems domain for later courses.

The first semester of this sequence focuses on the structure and execution of sequential software. These C-based projects revolve around binary executables structured in a simplified version of ELF (called Mini-ELF) and targeting the Y86 architecture [4]. Figure 1 shows a short description of each project, along with the specific learning goal for this phase.

The primary goal of this tool is to automate the delivery of feedback, which is an essential component of learning [9]. However, the quality and learning benefit of the feedback depends on several characteristics relating to how it is structured and given [13]. For instance, good feedback is goal-referenced and actionable, allowing the student to monitor their learning progress. In addition, good feedback is timely and user-friendly. Satisfying all of these properties is particularly challenging within the constraints of systems work that often relies on command-line tools. As such, one design goal of our system is to integrate the output from multiple tools into a single summary report that informs students regarding the tests their code passes, whether there are non-obvious concerns (such as memory leaks), and how their work will be measured for grading purposes. Since the students have immediate access to all of the tests, they get this feedback immediately *even though some tests are kept private to prevent leaking implementation hints.*

In addition to the intuitive goals of automating grading and feedback, our goals for this system are inspired by other work relating to motivation and persistence. Specifically, grades are a form of extrinsic motivation that reduces students' ability for self-regulation and control; as grades are perceived as either a reward or punishment, the net effect on students' intrinsic motivation is negative and detrimental to learning [11]. To counter this effect, we strove to structure our tests as an incremental approach based on the notion of proximal subgoals [1]. That is, with both unit and integration tests progressing through a series of closely related stages from simple to complex, successfully passing one test supports the students' expectation of passing the next.

3 TESTING FRAMEWORK

The testing framework is publicly available at *https://github.com/JMU-CS/c-test-framework*. The framework is designed for C99 on Linux, intended to be used in an academic course. It uses the Check[8] framework for unit testing, some custom Bash scripts for integration testing, and the Memcheck memory leak tool based on Valgrind[5]. We have attempted to clean up and generalize the code in this repository, but you may still find references to our course in these files. When downloaded a project directory would contain all the source files given to the student for the project. Some of these files will be "boiler-plate" templates that students will fill in with their solution. Included in this directory, is another directory called tests that contains the framework that we describe below.

This framework consists of unit tests for each function executed (in public.c and private.c), along with integration tests (in integrations.sh) that test overall output differences from running the student solution against a reference solution using diff on a given set of inputs. These tests form a testsuite that is then broken down into tests that are required for a particular grade. The sections below describe the unit tests, integration tests, the distribution and grading in more detail.

3.1 Unit Tests

Figure 2 shows sample code for creating a unit test according to the Check framework. The example here is setting up a test for a C-language function the students are asked to write for p0 that returns the sum of the absolute value of two numbers. This is a public test (i.e. one for which students can see the source code) and only tests the most straightforward case for this problem, adding two positive

Project	Description	Goal
`p0-intro`	Introductory set of C functions ranging from the absolute value add shown in this paper to more complex file manipulation problems	Master basic C programming
`p1-check`	Read in a Mini-ELF header and display it	Write standard Linux command-line programs and reinforce the idea that information is bits plus context
`p2-load`	Load Mini-ELF file into memory array	Reinforce the relationship between a compiled executable and the standard Linux/C memory model
`p3-disas`	Decode and disassemble machine code instructions	Reinforce assembly code concepts and the standard components of executable files
`p4-interp`	Simulate execution of machine code instructions	Reinforce assembly code and CPU architecture concepts, focusing on the execution of machine code instructions

Figure 1: Five programming projects used in our introduction to systems course

```
START_TEST (C_addabs)
{
    ck_assert_int_eq (add_abs(2,3), 5);
}
END_TEST
```

Figure 2: Example test case using the Check framework

```
START_TEST (A_sortarray)
{
    const size_t N = 4;
    int nums[] = { 5, 2, 11, 8 };
    int ref[]  = { 2, 5, 8, 11 };
    sort_array(nums, N);
    for (int i=0; i<N; i++) {
        ck_assert_int_eq (nums[i], ref[i]);
    }
}
END_TEST
```

Figure 3: More complex example test case using the Check framework

numbers. The name of the test starts with "C" indicating this test must be passed for the student to get a C grade. The test case including negative numbers in one or more of the two operands is a private test, also required for a C grade. Note that all of the tests have somewhat descriptive names so even failure of the private tests provides some feedback to students about what is wrong with their code. As an example, the name of the private test just referred to is `C_addabs_negative_ints`. Figure 3 shows a more complex test case.

3.2 Integration Tests

Integration tests are intended to test whole-program behavior and are written using input/output file pairs. Tests are specified in a configuration file, which contains one line for each integration test. The line specifies the name of the test (the output file must match this) as well as the command line intended to evoke the corresponding output.

```
int addabs (int num1, int num2)
{
    // BEGIN_SOLUTION
    return abs(num1) + abs(num2);
    // END_SOLUTION
    // BOILERPLATE: return 0;
}
```

Figure 4: Reference solution

```
int add_abs (int num1, int num2)
{
    return 0;
}
```

Figure 5: Distribution (boilerplate) version

The actual testing is handled by a Bash script that runs the program once per test with the specified command line and uses the `diff` utility to compare the actual output to the expected output. The script is designed to be generic enough to handle any project.

Because memory management is such an important part of coding in a low-level language, the script also runs each test in Valgrind/Memcheck to check for memory leaks.

3.3 Distributions

Scripts are provided for building project distributions to give to students. These scripts remove solution code and package up the project into a tarball. For convenient testing, it also saves a snapshot of both the distribution files and the full solution.

To mark solution code for removal, the instructor should surround it with `BEGIN_SOLUTION` and `END_SOLUTION` one-line comments. If the removal of the code would result in invalid code (e.g., no return value), a `BOILERPLATE` tag can provide alternative code for the distribution. Figure 4 shows one example of a reference function that has been marked prior to distribution, and Figure 5 shows how the same function appears in the generated file.

3.4 Grading

We also provide a sample grading script. To use this script, the instructor must collect student submissions using a specific file structure as described in the framework documentation. The instructor must also provide a configuration file with the relevant filename and path information for the reference solution, submission folders, and results.

When the grading script is run, it copies all of the student submissions into sub-folders in a temporary folder. It also copies the test files from the reference solution to prevent students from gaming the system by modifying the test files. It then builds and runs all tests on all submissions. The results are printed to standard output in a summary format, with overview test results for each student as well as information about compiler warnings, memory leaks, or insecure functions (e.g., `gets`). Figure 6 shows sample output from this script.

All unit and integration tests are labeled with a grade level. The lowest grade tests are the easiest to write code to solve and are the least dependent on other functionality. Also, to pass the lowest level tests does not require passing any of the upper level tests. The higher grade tests may be dependent on the lower grade tests and to get a higher grade, a student must also pass all of the lower grade tests. For example, to get a C the student's code must have passed all of the F, D and C tests that are given. This structure gives an overall road map to students of how to work on the project - starting with the lowest grade functionality and moving upward. As they pass each level of tests, this success gives them both confidence and a new incremental goal to pass the next level.

3.5 Student Use

Students primarily interact with the framework through provided Makefiles. They build their project using the default `make` command, or they can run the test suite using `make test`. Figure 7 shows sample output for an incomplete submission; the unit test output is provided by the Check library and the integration test output is from our script. Figure 8 shows sample output for a complete submission.

We encourage students to run the integration tests themselves outside of the test framework. The command line and input file(s) for all integration tests are listed in a configuration file so there is enough transparency for students to reproduce the tests. This enables them to iterate faster once they have identified tests that they are failing, and it enables them to run the test in an interactive debugger.

3.6 Docker Support

The framework has been designed to be platform-independent and has been tested on a variety of Linux platforms as well as macOS High Sierra and Windows 10 (using Cygwin). In addition, we provide Docker files so that the framework can be used on non-supported platforms. The resulting Docker containers can be used to build the project distributions, run the test suite, or start the program in an interactive debugging session using the GNU debugger, regardless of the host platform.

4 DISCUSSION

Below is a list of benefits we have experienced from using this automated framework with a brief discussion of each.

- **Having a built-in test suite forces students to read the project specification more carefully.** Many skim the description and don't spend time reading and thinking about it. Being confronted with failure when they expect success reinforces that they are working with a machine not a human ("machine as other"), and even tiny differences or misunderstanding can cause catastrophic failure. Details matter.
- **Having unit tests for the students to use increased their motivation, and subsequently their learning.** Students come with questions *before* the assignment is due as they are motivated to understand what is going wrong in order to improve their grade. In addition, the smaller unit tests allow them to focus on a specific goal and ask questions targeting that goal.
- **Scaffolded (grade-specific) tests give students specific, smaller goals and a road map through a large project.** Though not always apparent to students, the thoughtful labeling of the tests so that they are encouraged to work on the D tests first, then the C tests, etc. gives them a subtle road map on how to approach the large project. This support is then available for students who needed it without requiring individual graded milestones. It is a good middle-ground between short-term deadlines and a long-term project without interim support.
- **Providing some built-in unit and integration tests scaffolds students into creating their own tests.** Students seem to have very little idea of how to create test cases of their own (and particularly integration tests) when they arrive in this third semester course. Although some of this may just be unfamiliarity with the C language, providing example tests gives them a starting point and encourages them to create their own.
- **Having unit and integration tests encourages students to understand the difference.** Passing unit tests and then failing integration tests seems to highlight the importance of communicating the interface between pieces of software clearly as well.
- **The distribution of a testing framework facilitates exploration and learning beyond what we consider "necessary" to pass the class.** While some students will run `make test` and treat it like a magic oracle for the entire semester, other students dig in deeper to learn how it works, and a few will even tinker and experiment with it. When a test fails on a later project, the explorers and tinkerers have a greater understanding of how to begin debugging. This ends up performing as differentiated instruction based on interest and ability. Students can learn at the pace and depth that is appropriate for them.
- **Our test-based baseline grading scheme reduces student uncertainty about their grade.** This reduction in uncertainty reduces anxiety for most students and gives them an indicator of where they are in addition to how much time they will likely need to complete the project. It also allows

```
studentid1  C D D  25%: Checks: 20, Failures: 15, Errors: 0    Integration failures: 14    No memory leak found.
studentid2      X  (no submission)
studentid3  D D D  25%: Checks: 20, Failures: 14, Errors: 1    Integration failures: 14    No memory leak found.
studentid4  F D F  20%: Checks: 20, Failures: 16, Errors: 0    Integration failures: 14    No memory leak found.
studentid5  A A A  100%: Checks: 20, Failures: 0, Errors: 0    Integration failures: 0     No memory leak found.
```

Figure 6: Example grading script output

```
========================================
             UNIT TESTS
15%: Checks: 20, Failures: 17, Errors: 0
D_addabs:0: Assertion 'add_abs(2,3) == 5' failed [...]
C_addptr:0: Assertion 'ans == 5' failed [...]
C_factorial:0: Assertion 'factorial(1) == 1' failed [...]
C_isprime:0: Assertion '!is_prime(4)' failed
[...]
========================================
          INTEGRATION TESTS
D_hello         FAIL (see outputs/D_hello.diff for details)
C_goodbye       FAIL (see outputs/C_goodbye.diff for details)
[...]
No memory leak found.
========================================
```

Figure 7: Excerpt from test run for incomplete submission (some output omitted for brevity)

```
========================================
             UNIT TESTS
100%: Checks: 20, Failures: 0, Errors: 0
========================================
          INTEGRATION TESTS
D_hello                         pass
C_goodbye                       pass
[...]
No memory leak found.
========================================
```

Figure 8: Excerpt from test run for complete submission

the instructor to give quick feedback the day before something is due to prevent surprise zeros because of incorrect submissions.

- **Having some tests public and some private with more detailed error messages, enables more detailed feedback to students.** The detailed error messages let students and faculty know where student code is failing. Faculty can then spend more time on code inspection of individual submissions for style and approach if desired.

We have also observed some potential obstacles to learning introduced by our framework:

- **The framework initially confuses students because they don't understand the output, so they often have trouble getting started.** To address this, we accompany the framework with an extensive description of the framework and spend a lot of time in office hours working with any students who have difficulty understanding it. We also provide an online discussion forum for students to ask questions, and in our experience some of these questions are answered before we even see them by peers who learned the framework more quickly. Ultimately, we feel that mastering a non-trivial testing framework is an important experience in systems programming.
- **The framework reduces incentive for students to write their own tests even as it also scaffolds them into it better.** We feel a framework like this could be more appropriate at lower-level class work (our course is a second-year systems introduction course) before we really expect students to be able to rigorously test their own code. Future work could include adding hidden tests that are used for grading but aren't distributed to students in order to encourage more self-testing.

5 CONCLUSIONS AND FUTURE WORK

The most exciting conclusion from using this framework is the increased motivation and engagement by the students to dig deep and finish the projects. Anecdotal feedback from students is that they would not have worked so hard and gotten the details right if they had not known they did not yet have the grade they wanted. Additional anecdotal feedback is that by the end of the course the relationship of the project to the theoretical course content clicks and they are able to firmly grasp the theoretical concepts while studying for the final exam. The specification of specific tests for attaining each grade level, allows the introduction of a subtle road map to finish the project while avoiding the "hand-holding" of very short term graded milestones in earlier courses. In addition, these grade level tests introduce a mastery-model of sorts, encouraging students with short-term achievable goals and giving them confidence as they attempt more and more difficult pieces. As a result, we see more students perform better in later classes in the curriculum because they have mastered the basic material. This has reinforced the benefit of the learner-centered redesign of the curriculum.

In future work, we will investigate adding static analysis such as that provided by lint to the testing framework. This would expand the immediate feedback available to students and support them in writing cleaner and safer code. We will also experiment with expanding the projects to include optional parts that could differ in rotations over semesters. In addition, changing the output and formatting of the tests to provide more encouragement and direction to students is an option. An example might be, "Congratulations on passing test X, you should consider working on test Y." Trade-offs on autonomy vs. direction would need to be considered here, but in some cases this encouragement could be helpful.

REFERENCES

[1] Albert Bandura and Dale H. Schunk. 1981. Cultivating Competence, Self-Efficacy, and Intrinsic Interest Through Proximal Self-Motivation. *Journal of Personality and Social Psychology* 41, 3 (September 1981), 586–598.

[2] Don Blaheta. 2015. Unci: A C++-based Unit-testing Framework for Intro Students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 475–480. https://doi.org/10.1145/2676723.2677228

[3] Christopher Brown, Robert Pastel, Marika Seigel, Charles Wallace, and Linda Ott. 2014. Adding Unit Test Experience to a Usability Centered Project Course. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 259–264. https://doi.org/10.1145/2538862.2538964

[4] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective (3rd Edition)*. Boston.

[5] Julian Seward et al. 2017. Valgrind. http://valgrind.org/

[6] Michael S. Kirkpatrick, Mohamed Aboutabl, David Bernstein, and Sharon Simmons. 2015. Backward Design: An Integrated Approach to a Systems Curriculum. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 30–35. https://doi.org/10.1145/2676723.2677264

[7] Zachary Kurmas. 2017. MIPSUnit: A Unit Testing Framework for MIPS Assembly. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 351–355. https://doi.org/10.1145/3017680.3017747

[8] Arien Malec, Branden Archer, Chris Pickett, Fredrik Hugosson, and Robert Lemmen. 2014. Check: Unit Testing Framework for C. https://libcheck.github.io/check/

[9] National Research Council. 2000. *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*. The National Academy Press, Washington, DC. https://doi.org/10.17226/9853 Edited by John D. Bransford, Ann L. Brown, and Rodney R. Cocking.

[10] Linda B. Nilson. 2014. *Specifications Grading: Restoring Rigor, Motivating Students, and Saving Faculty Time*. Stylus Publishing, Sterling, VA.

[11] Richard M. Ryan and Edward L. Deci. 2000. Self-Determination Theory and the Facilitation of Intrinsic Motivation, Social Development, and Well-Being. *American Psychologist* 55, 1 (2000), 68–78.

[12] Michael Wick, Daniel Stevenson, and Paul Wagner. 2005. Using Testing and JUnit Across the Curriculum. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*. ACM, New York, NY, USA, 236–240. https://doi.org/10.1145/1047344.1047427

[13] Grant Wiggins. 2012. Seven Keys to Effective Feedback. *Educational Leadership* 70, 1 (2012), 10–16. Feedback for Learning.