IMPACT OF AUTO-GRADING ON AN INTRODUCTORY COMPUTING COURSE

Mark Sherman, Sarita Bassil, Derrell Lipman, Nat Tuck, Fred Martin
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01852
978-934-{1964, 3911, 1964*}
{msherman, sbassil, dlipman, ntuck, fredm}@cs.uml.edu

ABSTRACT

This project presents and assesses the impact of a new teaching tool that was deployed in an introductory Computer Science course. The tool was a web site that accepted student submissions for assignments, and automatically tested them for correctness. The students received feedback moments after submission. They could use that feedback to improve their work, and submitted again. We compared student performance on assignments against previous semesters, which used the same assignments, but with no automated feedback system. We observed that students, when using the feedback system, make more submissions per assignment, indicating that students were leveraging feedback to improve their programs.

INTRODUCTION

We introduced a new system for auto-grading student programming assignments in an introductory computing course, and assessed the impact of that system on teaching. This system simplified the logistics of the grading process and helped students by providing near-immediate evaluation of their assignments.

The assignments in this course were programming exercises written in C. The course typically has 40 to 60 individual programming exercises, and 40 to 60 students in a course section. The development of this system was informed by other state-of-the-art systems developed at other universities, such as BOSS [5], CourseMarker [3], and Web-Cat [2].

In the existing course, students are assigned programs approximately every lecture (multiple times per week). It is important to get feedback to the students as soon as possible [0], and this pace (coupled with the large enrollments of 50 to 60 students per section) creates a large workload for the instructor and the graders.

This is not only a logistical challenge, but also a pedagogical problem. Before this system, a student only saw feedback of an assignment after completing some number of successive assignments. This delay of feedback was detrimental to students, inhibiting their ability for informed iteration.

The system was designed to achieve these features:

1) Accept student code.
2) Run test cases on student code.
3) Display test results to the student nearly immediately.
4) Store the results of test cases for instructor (or human grader).
5) Store students code for manual review by instructor (or human grader).
6) Provide efficient interface for grader to review tests, read code, and generate feedback.

This is a common situation for institutions teaching computer science [4]. The above features are aligned with the desirable features described by Ala-Mutka [1]. The system described in this paper represents a secure, lightweight, and easy-to-use solution to this feature space. Specifically, this system is an online automatic assessment tool, which runs pre-made tests on student submissions, but also allows for easy human assessment of the code and the test results.

Technical Description of System

This system was initially developed to support the teaching of a "flipped" course, where students watch video lectures online before class to prepare for classroom questions and discussion. It also included online submission and grading of programming assignments, which were useful functions for traditional courses, as is examined in this paper.

The system was built using the Ruby on Rails [6] web application development framework. This framework allowed the application to be built rapidly, but also provides built-in automated testing infrastructure which has helped the application stay high quality and maintainable as it grew. The application followed standard Rails conventions. A PostgreSQL2 database was used to store most application state, although student submissions were stored on the file system.

A simple process for online submission of assignments was provided. Students were emailed authentication links that brought them to their list of assignments, and identified the students to the application. Assignments were submitted by uploading the programming code directly in their web browser. Both assignments requiring submission of a single source file and assignments requiring multiple files (submitted as a compressed archive file) were supported. The automated grading process began immediately when an assignment was submitted, giving students feedback within a few seconds. Students could attempt submissions multiple times.

In order to automatically grade student programs, submissions were compiled and run on the server. Allowing students to run arbitrary code on the server is clearly a potential security issue [4]. The system used a sandbox mechanism to prevent student programs from causing trouble. Five major techniques are used to isolate student programs from the rest of the system:

1) **Separate system user** - Each student program was run under a separate system user with minimal Unix permissions.
2) **Run in a "chroot"** - Student programs could only access specific, white-listed parts of the file system.
(3) **Resource limits** - The "setrlimit" system call was used to set limits on the use of a variety of resources, including RAM, child processes, and created file size.
(4) **Isolated working directory** - Each program was executed in a separate "tmpfs" file system, which ceased to exist when the grading process finished.
(5) **Watchdog timer** - A grading process was terminated if it lasted more than five minutes.

This sandbox mechanism did not provide perfect security, and may be vulnerable to a clever student intentionally trying to defeat it, though we have not yet identified any

such vulnerability. It did perform adequately at preventing the grading server from being disrupted by common student mistakes, like infinite memory allocation loops, without the need for exotic isolation systems, virtual machines, or additional servers.

Writing Tests

In preparation for student submissions of assignments to the system, the instructor writes a set of test scripts, which automatically evaluate specific aspects of students' programs. A test may evaluate, for example, that given specific input, the program generates proper output; it may check whether certain required functions are provided by the student's program; it can determine whether the program exited successfully or crashed; etc.

Tests can be written in any language. The only requirements for tests are that they conform to the Test Anything Protocol (TAP) [7]. With TAP, a test outputs an indication of the number of tests that will be run, e.g., 1..4 for four tests, followed by a line of output for each test, which indicates whether the test, e.g., test #2, succeeded (ok 2) or failed (not ok 2). The tests are otherwise free to provide input in any form to the program under test and to retrieve output via their standard output mechanism or via a file. Tests are also free to provide additional output, such as a description of the test to be run, or student's and expected output, when a program does not yield the correct results.

Given the flexibility of this testing harness, it is possible to run some interesting tests on students' programs to let them know that they are correctly accomplishing their program's goal. While beginning study of heap allocation using the malloc and free functions in C, for example, we used a test that compiled the student's program into an object file, and then used the *objcopy* utility to replace their main function with one of the test's own, and to replace their calls to malloc and free with calls to alternate function names which were defined in the test. In a simple, early heap allocation assignment, students were to write a program that allocated one integer of heap storage, assigned the number 6 into that storage, and then freed the allocated memory. The functions in the test which replaced *malloc* and free were then able to ensure that *malloc* was being called with the correct size, that the correct address was passed to free, and that the value 6 had in fact been put in the correct memory location by the student's program. Similarly, the replacement functions were able to test that the student's application correctly handled an out-of-memory condition as indicated to their program by a NULL return from *malloc*.

As we gained more experience writing tests for this system, we found that we could easily run student programs under *valgrind* with varying options to detect different types of student errors. We were able to use *nm* to look at their object file to confirm that they were providing the specified functions, and even implement unit testing of those specific functions. For example, in a bubble sort assignment, the students were required to implement a *swapValues* function to swap two values (for practice passing pointers to integers), and a *bubblesort* function that was required to use the *swapValues* function. The tests for this program were able to unit test the students' *swapValues* functions to confirm correct operation, count the number of calls made to *swapValues* to ensure the student's algorithm was correctly implemented, and test the entire program for correct output given various inputs.

The simplicity of this platform allowed for any test to be written than can be realized through another program, making the testing mechanism more flexible and

robust than those used in other C-language studies, which largely depend on comparing student code and results against a reference implementation, such as Sterbini and Temperini [8] did, among others.

METHODOLOGY

Analysis was performed on historic course data, with IRB-approved protocols to de-identify data and protect participants. The data included grades the students received on their submissions to assignments, as well as the submissions themselves, which were C source code files.

Two forms of analysis took place: (1) Aggregate analysis, tallying all submissions per assignment, per course offering, resulting in numbers of student submissions in each assignment, and (2) fine-grain submission rate analysis, where the distribution of submission rates among students was visible.

The analysis used data from seven offerings of the same introductory computing course, spanning four semesters and three instructors. Students with zero submissions, instructors, administrators, and other non-students were removed from the data. Assignments that were not programming assignments were also removed from the data, such as in-class paper exercises, quizzes, and tests. The data only represented participating students in the courses, and take-home programming assignments. All sections of the course used a common, core set of assignments. Assignments that deviated greatly from the common set, such as those with novel specifications or dependant on new concepts, were removed from the data.

RESULTS

Aggregate analysis showed a substantial increase in the number of submissions students made when using the auto-grading system. Shown in Table 1, course sections M1-M4 were entirely manually graded, and sections A1-A3 used the auto-grading system. The numbers of total submissions made, participating students, and assignments were divided to create the "Submissions per Student per Assignment" descriptor, which indicated the general rates of submissions per assignment in each section. A value of 1.0 would indicate that every student made exactly one submission for every assignment. Sadly, many students do not submit for every assignment, bringing down the average submission rates.

Course Section	Submits/Student/Assignment	Average	Difference
M1	0.9706	0.9831	1.829x
M2	0.9974		
M3	0.9898		
M4	0.9747		
A1	2.0387	1.7982	
A2	1.7016		
A3	1.6544		

Table 1: Submission rates among course sections

We also used a finer analysis technique, where the average submission rate per assignment was computed for each student. With this, we could see the distribution of submission rates within the course sections. In Figure 1, we can see the changes in that distribution across course sections as a histogram. Like above, sections M1-M4 were manually graded, and sections A1-A3 utilized the auto-grader. The four category bars in each cluster represent four ranges of submission rates, where a rate of 1.0 would be in the second category.

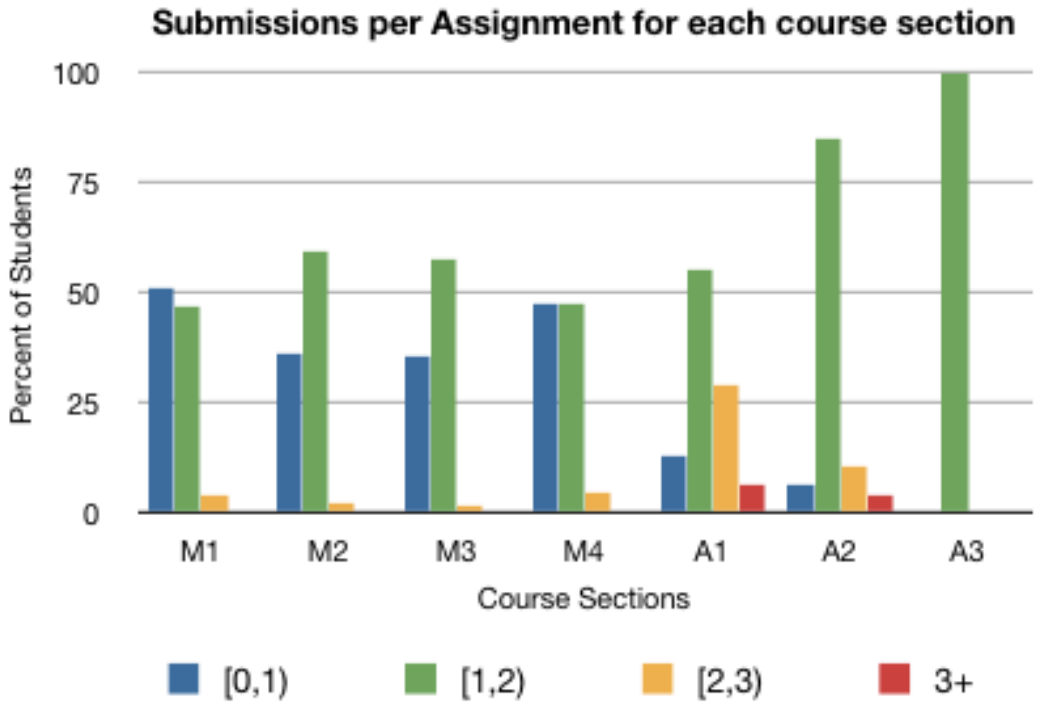

Figure 1: Histogram of how many students had similar submission rates

CONCLUSIONS

As seen in Table 1, the standard deviation of the Submission/Student/Assignment rates of the manually graded sections was 0.013, indicating that, despite spanning multiple semesters and multiple instructors, submission rates before the introduction of the auto-grading system were consistent. The course sections that used the auto-grading system showed higher submission rates than the manually graded sections. This increase was significant, showing that students, on average, made nearly twice as many attempts on an assignment when using the auto-grading system.

The histogram in Figure 1 shows all seven sections, and a course distribution of submission rates in each section. Note that the courses that were manually graded, M1-M4, showed a large portion of the students submitting, on average, less than once per assignment for every assignment. In those sections, almost none of the students had averages above two submissions per assignment. In M1 specifically, the lowest submission rate was seen in over half of the students. In the sections that used auto-grading, the first class of low-submission rates immediately falls off, with a greater

number of students submitting more than once, and some students submitting more than twice. Two sections, A1 and A2, show a portion of the class had an average submission rate of over three, which is partially attributable to students submitting the same code repeatedly, and/or trying to depend on the auto-grader as their only source of compilation and error recognition.

In total, this figure indicates a healthy change in mindset towards submitting, where students are using the system to help them make one or two additional iterations on their program. The lack of significantly higher average submission rates indicates that students are not trying to abuse the system, by over-depending on it, or using it as a compiler.

FUTURE WORK

There is a wealth of data from this study remaining to be analyzed, including looking at the qualitative performance of student work. In using this system, many tests were written by the instructors, which include the feedback that the students see. There may be relationships between the types of tests used, the quality and kind of feedback provided to the student, and student performance. This avenue of study is promising.

This system will also be deployed in an upcoming semester in a "flipped classroom" graduate course, which may produce data in a different context, but still with connections between tests, feedback, and the student.

REFERENCES

[1] Ala-Mutka, K., A survey of automated assessment approaches for programming assignments, *Computer Science Education,* 15, (2), 83-102, 2005.

[2] Edwards, S., Perez-Quinones, M., Web-CAT: automatically grading programming assignments, In *Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITiCSE '08),* 2008.

[3] Higgins, C., Gray, G., Symeonidis, P., Tsintsifas, A., Automated assessment and experiences of teaching programming, *ACM Journal on Educational Resources in Computing,* 5, (3), 5, 5, 2005.

[4] Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O., Review of recent systems for automatic assessment of programming assignments, In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*, 86-93, 2010.

[5] Joy, M., Griffiths, N., Boyatt, R., The BOSS online submission and assessment system, *ACM Journal on Educational Resources in Computing,* 5, (3), 5, 2, 2005.

[6] Ruby on Rails, http://rubyonrails.org/ , retrieved November 15, 2012.

[7] Schwern, M., Lester, A., Documentation for the TAP format, 2003, http://search.cpan.org/~petdance/Test-Harness-2.64/lib/Test/Harness/TAP.pod , retrieved November 15, 2012.

[8] Sterbini, A., Temperini, M., Automatic correction of C programming exercises through unit-testing and aspect-programming, In *Proceedings of the 2nd International Conference on Educational Information Systems, Technologies, and Applications (EISTA '04),* 6, 2004.