IMPACT OF AUTO-GRADING OF AN INTRODUCTORY COMPUTING COURSE

Person Person, Author Author, Author McAuthorson,

Department of Computer Science

Institution

City, ST 00000

111 111-1111

{email, email, email, email}@domain

**ABSTRACT**

This project presents and assesses the impact of a new teaching tool that is being deployed in an introductory Computer Science course. The tool is a web site that accepts student submissions for assignments, and automatically tests them for correctness. The students receive feedback moments after submission. They may use that feedback to improve their work, and submit 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, and have qualitative evidence of students leveraging feedback to improve their programs.

**INTRODUCTION**

We introduce a new system for auto-grading student programming assignments in an introductory computing course, as well as an assessment of the impact of that system on teaching. This system simplifies the logistics of the grading process and can help students by providing near-immediate evaluation of their assignments.

The assignments in this course are 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 [$BOSS], CourseMarker [$COURSEMARKER], and Web-Cat [$WEBCAT].

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, 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. Presently, a student only sees feedback of an assignment after completing some number of successive assignments. This delay of feedback is detrimental to students, inhibiting their ability for informed iteration. If we can provide students with immediate feedback on their work, we can solve the pedagogical problem and use staff time more efficiently.

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 [$IHANTOLA]. The above features are aligned with the desirable features described by Ala-Mutka [$]. 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. Bottlenose also includes online submission and grading of programming assignments, which turns out to be a useful piece of functionality even for traditional courses.

Bottlenose is built using the Ruby on Rails [$RAILS] web application development framework. This framework has allowed the application to be built rapidly, but also provides built-in automated testing infrastructure which will help the application stay high quality and maintainable as it grows. The application follows standard Rails conventions. A PostgreSQL2 database is used to store most application state, although student submissions are stored on the file system.

A simple process for online submission of assignments is provided. Students are emailed an authentication link that brings them to their list of assignments and identifies them to the application. Assignments are 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) are supported. The automated grading process begins immediately when an assignment is submitted, giving students feedback within a few seconds. Students may attempt submissions multiple times.

In order to automatically grade student programs, submissions are compiled and run on the server. Allowing students to run arbitrary code on the server is clearly a potential security issue. Bottlenose uses 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 is run under a separate system user with minimal Unix permissions.

2) **Run in a “chroot”** - Student programs can only access specific, white-listed parts of the file system.

(3) **Resource limits** - The “setrlimit” system call is 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 is executed in a separate “tmpfs” file system, which ceases to exist when the grading process finishes.

(5) **Watchdog timer** - A grading process is terminated if it lasts more than five minutes.

This sandbox mechanism does 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 does 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] (TAP). 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 allows 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 [$] 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) small case studies, where a sample of students’ individual paths through particular assignments were qualitatively observed and described.

The aggregate 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.

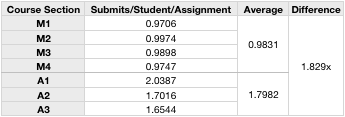


Table : Submission rates among course sections

The standard deviation of the S/S/A rates of the manually graded sections was 0.013, indicating that, despite spanning multiple semesters and instructors, submission rates before the introduction of the auto-grading system were consistent.

**Increase in Re-Submission Rate**

**Case Studies**

**CONCLUSIONS**

**DISCUSSION**

**REFERENCES**

BOSS, University of Warwick. http://www.dcs.warwick.ac.uk/boss/

[BOSS] 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.  
  
CourseMarker, University of Nottingham. http://www.cs.nott.ac.uk/~smx/PGCHE/courseMarker.html

[CourseMarker] 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.  
  
Web-Cat, Virginia Tech Institute for Distance and Distributed Learning, with NSF support. http://wiki.web-cat.org/WCWiki/

[WebCAT] 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.

Test Anything Protocol http://search.cpan.org/~petdance/Test-Harness-2.64/lib/Test/Harness/TAP.pod

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.

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