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**

One paragraph overview of the paper.

**INTRODUCTION**

We propose a new system that simplifies the logistics of the grading process of student assignments in our Computing I course in the Department of Computer Science. In addition, the proposed system can help the students by providing a “just-in-time” 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. The development of the new system will be upon results from related 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 work more or less 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 a pedagogical problem. Presently, a student only sees feedback of assignment #1 after completing assignments #2, #3, or more. If we can provide students with immediate feedback on their work, we can solve the pedagogical problem and use staff time more efficiently.

Specifically, it would be nice if we can state clearly this list  
of features that are provided by Bottlenose:  
  
1) Accept student code.  
2) Run test cases on student code.  
3) Display "just-in-time" test results to the student.  
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.  
  
Also, we should stress the fact that, in this introductory computing  
course, students are assigned programming exercises multiple times per  
week, and pedagogically, it is important to provide 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 presents not only a logistical challenge, but a pedagogical problem  
as well, since a student only sees feedback of assignment #1 after  
completing assignments #2, #3, or more.  
  
Providing students with immediate feedback on their submitted assignments  
(cf. number 3 in the previous list of features), can solve the pedagogical  
problem and use staff time more efficiently.

**Background**

Boss is a tool for online submission, grading and course management. It is  
not a Computer-Assisted Learning (CAL) tool. Therefore, Boss should not  
affect students' learning experience. In our project, we aim to solve the  
pedagogical problem of providing to students feedback of assignment #1  
after completing assignments #2, #3, or more. Hence, our goal is to affect  
(enhance) students' learning experience by providing them with immediate  
feedback for correctness.  
  
Tests for an assignment might be student-accessible or not (flag to be set  
when editing tests).  
  
Two kinds of tests are possible in Boss:  
- Random test: generates a random number between two values rangemin  
(lower bound) and rangemax (upper bound) to be specified by staff.  
- Constant test: generates a constant result.  
  
Staff should manage students - add students to modules (many assignments  
per module - many tests per assignment)  
  
Four different roles can use Boss:  
- student  
- marker (or grader)  
- staff (i.e., instructor, he/she manages students and markers)  
- admin (manages Boss itself)  
  
University ID numbers are used for usernames  
  
Boss2 system is based on an MVC architecture (Model-View-Controller). This  
architecture is appropriate for web-based applications: a clear separation  
between the representation of information and the user's interaction with  
this information through user interfaces.

**Technical Description of System**

Our web application, Bottlenose, 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, especially against a clever student intentionally trying to break it. Nevertheless, it has worked reasonably well at preventing the grading server from being disrupted by common student mistakes like infinite memory allocation loops.

**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, for example, students were to write a program that allocate one integer of heap storage, assigned the number 6 into that storage, and then free 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, in addition to testing the bubble sort program for correct output given various inputs.

**METHODOLOGY**

Can use a lot from the IRB request. Sections may be spun off to include Research Participants, Analysis Protocols, etc, as needed.

**RESULTS**

Bam! Graph and stuff.

**Increase in Re-Submission Rate**

**Case Studies**

**CONCLUSIONS**

**DISCUSSION**

**REFERENCES**

BOSS, University of Warwick. http://www.dcs.warwick.ac.uk/boss/  
  
CourseMarker, University of Nottingham. http://www.cs.nott.ac.uk/~smx/PGCHE/courseMarker.html  
  
Web-Cat, Virginia Tech Institute for Distance and Distributed Learning, with NSF support. http://wiki.web-cat.org/WCWiki/

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