A Web-based Automatic Program Grader

Don Colton
don@colton.byuh.edu

Leslie Fife

Andrew Thompson

School of Computing
Brigham Young University Hawaii
Laie, Hawaii 96762 USA

ABSTRACT

The ability to program is one of the core tools used by computer scientists, and programming proficiency is a recommended requirement for ABET accreditation. In our experience, students learn programming skills best by writing many programs, ranging from simple to complex. Overworked teachers can be dismayed by the prospect of grading still more programs per student as well as teaching introductory classes with large enrollments. The automatic grading approach offers substantial advantages and opportunities, but also some challenges. We present WebBot, a web-based automatic grader for computer programming assignments. This program is an expansion of GradeBot, an automatic program grader used for several years. This newest version of GradeBot introduces a web-based interface. GradeBot evaluates student programs written in any of several languages, including C, C++, Java, Perl, Python, Tcl, and MIPS assembler. Guidance for similar projects is provided through a discussion of the development and use of GradeBot and WebBot.

Keywords: automated grading, grading, courseware, CS1, CS2, web-based

1. INTRODUCTION

When intermediate and advanced students in Computer Science are given one programming assignment each week throughout the semester, they are generally successful. However, when novice programming students in CS1/CS2 were assigned programs at the same pace, the results were not good. Students experiencing difficulty at this pace often gave up in frustration or acquired too much "unauthorized help," thus failing to learn the material. This has an obvious negative effect on retention and program completion.

We felt that inexperienced students were not successful with the pace of one program per week because it forced them to learn and demonstrate too much new material per program. Instead of weekly assignments fully demonstrating a new concept more frequent and smaller steps needed to be assigned and graded. In addition, these smaller steps needed to provide rapid feedback to allow the students to progress quickly from one step to the next. A change was made to better support the students by assigning and grading four or five smaller programs per week. While this is the right thing to do for student learning it has obvious drawbacks. The resulting grading burden on instructors and teaching assistants was the problem for which GradeBot became the solution.

The thesis of the GradeBot project is that student learning in introductory programming classes can be effectively facilitated by the use of an automatic program grader supporting multiple programming assignments each week.

1.1 Additional Motivation

Students in the introductory programming sequence are exposed to and taught the tools of a computer scientist. One of these important tools is the ability to program (Chang et al, 2001). Our primary motivation was to support students learning by breaking the

learning process into smaller steps in a supportable manner. Without learning to program well, a student is at a clear disadvantage throughout the curriculum. Other motivations include:

- Paradigm Shift: It is important to note that automatic grading offers but does not require a complete paradigm shift from traditional grading. There are two potential differences. (1) With automatic grading the student may be allowed to turn in the assignment many times without penalty, with the automatic grader evaluating each one quickly and patiently. Credit is only granted when the student program works. (2) It is still possible and may be desirable to involve a human grader to ensure a program was written as specified, including stylistic requirements. However, such involvement is not required.
- More Programs per Student: Automatic grading allows a move from 5 to 10 programs per student per semester toward a target of 50 programs per semester. Rather than the steep learning curve of one program for each topic, one might have many more programs, resulting in a more gradual learning curve.
- Faster Response to Students: With an automatic grader in place, students are able to submit their lab work at any time and find out immediately whether it was accepted.
- Objective (not Subjective) Grading: When student work is graded as a batch after the due date, teachers frequently give partial credit for incorrect work depending on how close to correct they feel it is. But "how close" is a subjective judgment, whereas "works or fails" is an objective judgment. Because students can resubmit incorrect work before the due date, and know when they are done, it becomes reasonable to use objective, all-or-nothing grading without partial credit, which effectively requires perfect programs.
- Students Debug Their Programs: Instead of showing or telling students where their program is wrong, students are simply given the test case led to their failure. They must figure out why their program failed and how to fix it. This is more true to life and provides greater learning.

2. GRADEBOT BACKGROUND

The web-based automatic program grader, WebBot, is built on and extends GradeBot. In this section we provide an overview of GradeBot. We then discuss the changes provided by WebBot. A treatment of GradeBot is available in Colton, Fife, and Winters (2005).

2.1 The History of Automatic Grading

Automatic program grading is not a new idea. What follows is a brief overview of the work in this area.

The earliest report of automatic programming was published in the CACM by Hollingsworth (1960). Later Forsythe and Wirth (1965) were using an automatic grading program in introductory programming courses in Algol. A significant drawback of this early system was that routines had to be written for each programming problem, and then the system was recompiled. However, the system provided random test data and checked student performance. Students included the appropriate procedure cards in order to have their programs graded. Forsythe and Wirth recommend the use of automatic program graders. Their system, while primitive, is a good example of the possibilities of automatic program grading.

BAGS (Hext and Winings, 1969) (Basser Automatic Grading Scheme) from the University of Sydney was a later system. Their system accepted work in three languages and tested the user programs with two data sets. The system gave 1 point for each of 5 activities: successful compile, complete run, data set 1 correct, data set 2 correct, and time sufficiently short. The program also penalized a student for each submission after the first. The ability to compile, run and test programs without human intervention is an important part of automatic grading. The philosophy of WebBot, however, differs from BAGS. With BAGS, you could get points for submitting the wrong program if it compiled and ran. You also get points for a program that works on some data, but not all.

Kassandra (von Matt, 1994) was an automatic grader used in the early 1990s. Kassandra would test according to two test cases, and give credit if both answers were correct. Kassandra also had the ability to provide students with a list of completed assignment.

Unfortunately, the two test cases were static. This can lead to issues with cheating and hard-coding results.

Finally, the ***submit*** program (Harris, Adams, and Harris, 2004) in use at James Madison University in 2004 shares some of the same philosophies with WebBot. For example, student programs must be precisely correct to receive credit, the only penalty being late submission. While submissions generate a report, it is unclear if a complete schedule for assignments done and pending can be created automatically. In addition, ***submit*** is a command line utility and not a complete automatic grading suite. Many potential instructor and student features are not present.

2.2 Grading Model

GradeBot works by comparing the behavior of a student program to a defined standard. The behavior consists of the outputs that are produced by the student program. If the student program performs as required, it is declared to be correct. If the student program fails, GradeBot can only identify the discrepancy in the student program output.

- Program Submission: Students submit their programs as source code in any of the target languages that the teacher permits. Supported languages include C, C++, Java, Perl, Python, Tcl, and MIPS assembler (SPIM). Once the program compiles cleanly, a series of zero or more tests are performed.
- In the original model, each test followed a standard in, standard out evaluation model. Under the current model standard in and standard out are interleaved as a dialog, using Expect (Libes, 1995) to verify that ins and outs happen in the right order.
- Test Cases: The original concept was to provide a few hand-made, hand-verified pairs of files for each test case. One file would be the input and the other file would be the output. The input file is fed into the student program. The output results are collected. Finally the collected results are compared with the desired output. This is repeated for all test cases.
- Failure Revealed: For instructional purposes, if there is a discrepancy between the desired output and the actual output, the failed test case is revealed to the student. This allows the beginning students to debug their own programs. It also avoids most cases of students protesting that their program was actually right. A counter-example serves as an effective proof.
- Helpful Comments: As much as possible, messages indicating the error cause or location are provided. Examples are: "Your first error is on line 5 of your output," "Please check your spacing," or "Please check your punctuation" Both the produced output and the correct output are also provided for comparison. Ideally the system would point out the place where the student program was wrong, rather than the place where the output was wrong. Humans can often do this, but it is beyond the capabilities of this system.
- Infinite Loops: Infinite loops were a problem. To deal with this, a timed execution facility called timed-run was used. It was already present on our Linux system as part of the **expect** package (Libes, 1995). Not foreseen were infinite loops with print statements nested inside. The first occurrence was a program that generated 100,000 identical lines of output before it timed out. It took an hour to email the results to the student. Two measures were adopted to mitigate the infinite loop print problem. First, before emailing, identical lines were eliminated. When there were three or more lines that were identical, only the first would be returned, followed by a statement such as "the next 183 lines are the same." In addition, the size of the desired output was used as a guide for what was reasonable, and outputs that were too much longer were simply truncated. The important thing was to give the student a good view of his output without falling into an infinite output.
- Program and Machine Crashes: Core dump files are created by failed compilations. These took up a large amount of disk space. A nightly "**cron** job" was set up to remove all core files within the testing directory tree. One or two clever and motivated students have found ways to crash the server, but they have been proud of their achievements and have been willing to accept acknowledgement for their cleverness. They have not been an ongoing source of annoyance. In seven

years there has been no need to deal with this.

- Creating New Labs: In order to keep programs from becoming too well known it was important that lab creation be simple. Although it was anticipated that new labs would be created frequently, in fact only a few new labs are created each year, mostly in response to new learning objectives rather than to avoid student cheating. We discuss cheating later.

2.3 Grading Engine

With the original grading model, students could in n tries discover all n test cases being used, since there were a finite number, and n was generally small for hand-verified test data.

- Plug-in Test Modules: To get beyond a small number of hand-verified test cases a plug-in was created to generate random inputs and matching outputs to test the student program. The plug-in ran each time it wanted input. The random number generator would create appropriate input. The input was then saved for the student program and also processed by the plug-in program. Each time the plug-in generated output, it was saved for comparison against the student program. An explanation of the random input generation together with a complete and annotated example can be found in Colton, Fife, and Winters (2005).
- Interactive Dialogue: Over time, the instructor was occasionally confronted by examples of student code that worked well enough for GradeBot but was still wrong. One typical example of this would be a program to ask for a number, read it in, add one to it, and print the result. The student program could instead read in the number, add one to it, and THEN ask for the number and print the result. Using standard in and standard out destroyed the interleaving sequence, the "dialogue," between input and output. All inputs could be read first, and then all outputs created. But the intention of the instructor was to have inputs and outputs interleaved.
- Longer Outputs: With computer-generated test files, it became practical to have longer input and output files. When all inputs and outputs were hand-generated and hand-verified, there was a strong tendency to keep things short and simple.

A major overhaul of GradeBot was conducted to get away from the batch input/output model, add the plug-in and allow for the longer running times of larger input. An interactive dialogue model was adopted for most program grading. Instead of comparing a whole output file, the student program outputs were verified one line at a time, as they were generated. Similarly, the inputs were provided one line at a time as they were needed. With this improvement, the student could be forced to prompt for input before actually reading the input.

An unexpected benefit of this approach was the fact that infinite printing loops were no longer a problem. At the first sign of trouble, the student program was terminated and the remaining dialogue was modeled for the student. Only the first error line was reported.

3. GRADEBOT TO WEBBOT

The original version of GradeBot had no visible user interface. All interaction with GradeBot occurred via email. A program was submitted via mail and responses came via email. The email facility was typically accessed from within **emacs**. While this interface was sufficient for experienced students, the time to teach novice students to use the system was two to three weeks. Novice students would have to learn to use **emacs**. Then, they would need to learn how to switch between edit and submit modes within the editor to submit labs and read responses.

A discussion began on adding a graphical user interface to GradeBot. There were several goals. The first goal was a graphical interface that a novice student could start and use immediately. This would allow the course instructor to focus on the course content and not the tool being used. Second, this interface needed to be available from anywhere. GradeBot was not limited to use in the departmental computer labs. A student could access GradeBot from anywhere they have email access. To maintain this same level of access, the upgraded GradeBot was developed as a web-based tool, WebBot.

A series of screen shots show the use of WebBot. In Figure 1 we see the login page.

Students also have a link if they forget their password. All students enrolled automatically have an account.

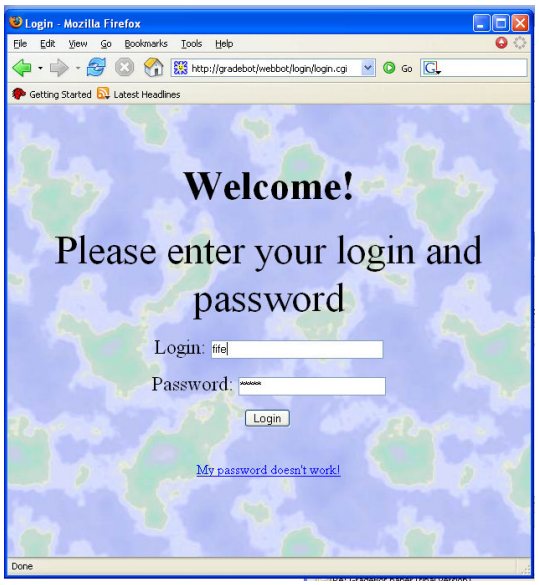

FIGURE 1: WebBot Login Page

In Figure 2 we see the WebBot frame enabled workspace. The workspace is divided into two regions.

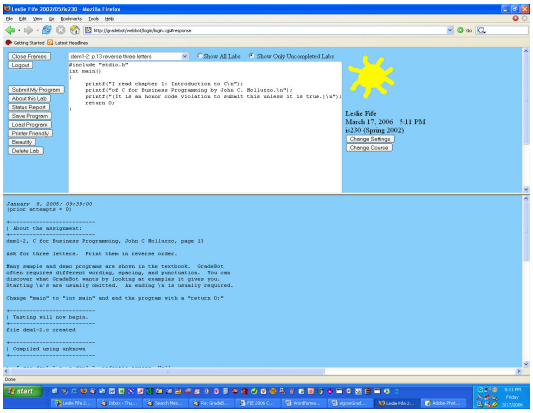

Figure 2: Frame Enabled Workspace

We see in Figure 3 that the non-frames version has a slightly different layout, but the same capabilities. The ability to switch between frames and no-frames gives the choice of preference to the user.

In the top portion we have a drop down selection of labs, buttons for control and a text window for entering program code. This code can be saved and loaded, so work can be saved between sessions. A beautify (pretty print) button formats the code for readability.

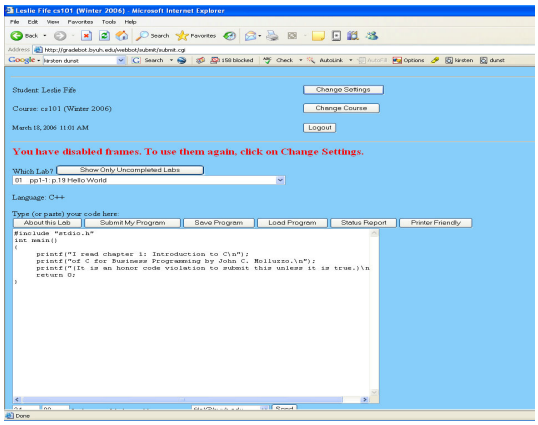

Figure 3: WebBot Non-Frames Workspace

At the bottom of the workspace is a message area. Responses from WebBot are provided in this space. A student can also review their entire course performance by an appropriate query to WebBot. This grade report is provided in the workspace message area. An example is shown in Figure 4. We can see in the design of WebBot that the interface functionality remains very simple.

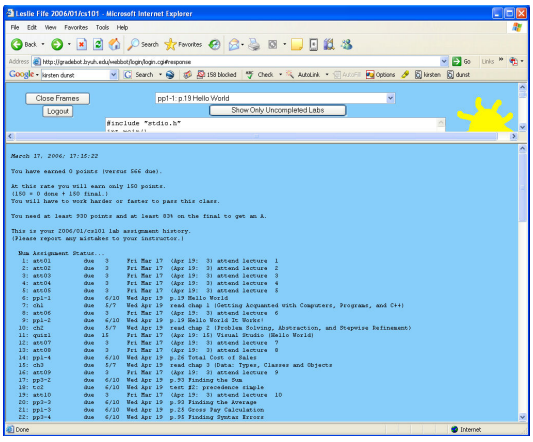

Figure 4: Student Status Report

The decision to keep the graphical interface simple was intentional. One problem with many IDEs is their complexity. It can be challenging to learn a complex IDE while also learning to program. When students are frustrated the source of the frustration can be the practice of programming or it may be learning the new tool. WebBot has a simple settings page, to change passwords. This is shown in Figure 5.

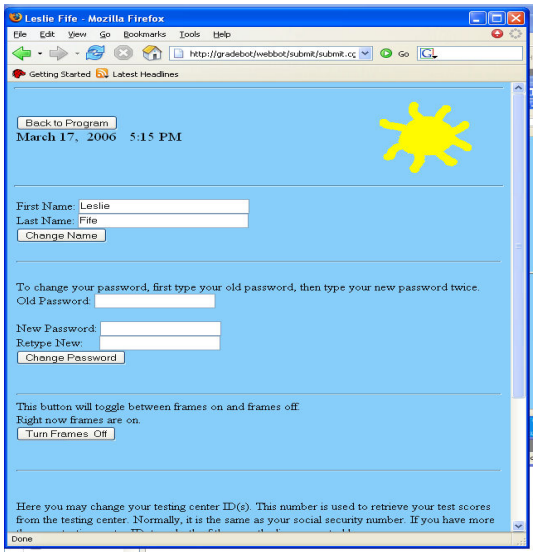

Figure 5: WebBot Options Page

WebBot also allows multiple courses to use the system. A student can enroll in multiple classes and switch between them. Students may also review past courses in which they have been enrolled. The course page is shown in Figure 6.

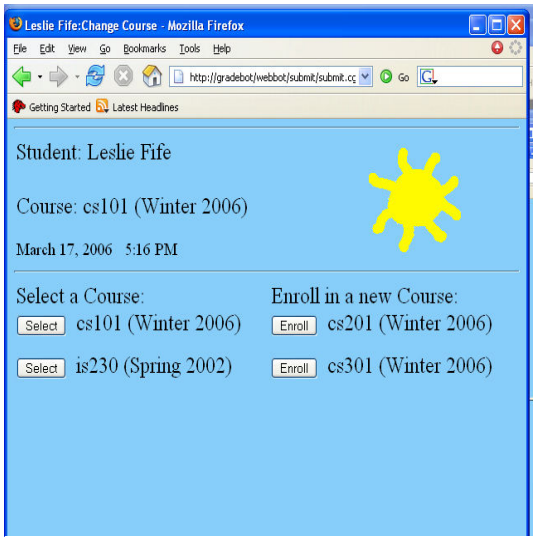

Figure 6: Course Selection Page

The web-based automatic program grader, WebBot, is built on and extends the earlier GradeBot. GradeBot was in use for more than four years before the introduction of WebBot. WebBot has now been functioning for three years. Two issues remain with the use of WebBot. First, students know how to program but do not know how to use the tools of programming, such as IDEs, editors and compilers. This is due to WebBot hiding the details of these tools from the student. While arguably appropriate for a novice learning to program, this is not acceptable for more experienced students. Second, the possibility of cheating exists. Both of these will be discussed.

4. WEBBOT DISCUSSION

4.1 IDE Introduction

The inability of students to use the tools of programming in an Integrated Development Environment (IDE) became evident immediately. Students in the course following CS1 did not know the steps necessary to edit, compile and run a program outside of WebBot.

This turned out to be a reasonably simple problem to address. The CS1 course was modified to introduce IDEs to the students. However, this introduction was delayed until late in the course. By the time IDEs were introduced, students were already comfortable writing and testing simple programs. At this point, the complexity of the IDE and the complexity of programming are separated in time. This has been successful.

4.2 Cheating

Some students were able to complete the labs but were still unable to perform on programming quizzes and tests given in class. Interviews with the department-provided tutors revealed that the students were helping each other, although explicitly forbidden.

There seemed to be two distinct elements contributing to the behavior. First, students seemed less upset about cheating in their interactions with a machine than they would in their interactions with a human. Second, as demonstrated by the 2001 GRE CS Subject Test cheating scandal (Pendell-Jones, 2003), in some cultures there is a strong us-versus-them mentality between students and teachers. Students are culturally expected to assist each other, even in defiance of instructor mandates. This cultural issue was more difficult to address.

To identify cheaters, GradeBot incorporated a complete history of all lab work ever submitted by students. When a new student program is submitted, it is compared with the

database. If a match is found, an incident report is emailed to the instructor. The incident report details the "miraculous" fact that two programs were identical.

The initial result was a lot of email. For simple labs, or for labs that represented only a small change from sample code given in the textbook, the odds of duplicate programs were quite high. This was also true for programs that were explained thoroughly in class by the instructor or in the lab by the tutors. Not all of this activity is cheating.

The next step was to look at the predecessors to any code match. For each match, the miracle report was modified to list all the previous identical submissions that had been received. If many students shared the same code there was generally an acceptable reason. If only one or two students shared the same code, it was more likely to be cheating. One incident was cause to be cautious, but did not provide enough evidence to convict.

The next step was to modify the report to include past incidents of identical code involving that student. This turned out to be very helpful. When student A had code that was like that of student B on one assignment, and like that of student C on another assignment, and like that of student D on yet another assignment, it could be attributed to the fact that there were a limited number of common ways to write the program. But if student A had code like that of student B on quite a few labs, this indicated a strong level of collusion.

We concluded that technical means could detect simple forms of copying, but effective police action was not practical because of the cultural desire to work together and the ease with which students could modify their copied work just enough to avoid being caught. It became easier to quit trying to directly control cheating on the labs and to allow students to work together. We rely on testing in a controlled setting to determine who has learned the course material. A large share of the final grade now rests on in-class tests. Students are explicitly permitted to do their lab work together, but are reminded that one important goal is the learning they will need to demonstrate on the in-class tests.

4.3 Test First Approach

There is a test-first philosophy that suggests students should enumerate all the possible or reasonable test cases before writing the program. We agree with this approach once students have gained a basic fluency in programming, but we believe starting too early with the test first approach can lead to the "paralysis of analysis" where you cannot learn to ride a bike because you cannot figure out all the things that could happen, and how to respond to each of them. Instead, we favor the approach that says: get on the seat and start pedaling; later you can think about what you are doing.

5. CONCLUSIONS AND FUTURE WORK

WebBot handles an average of 325 students per year, each submitting roughly 290 lab assignments to complete 25 labs per class, mostly in the CS1/CS2 courses. Table I shows actual statistics for three courses in 2004 and 2005. These courses are CS1, CS2, and Algorithm Analysis (CS 301). It has been used with a variety of student programming languages, including C, C++, Java, Perl, and MIPS (in the computer organization / architecture class). For a detailed example of its use, see the Appendix in Colton, Fife, and Winters (2005).

Instructors are very pleased with this tool, and desire to see it continued, but they are not totally satisfied due to some of the tradeoffs. Because the instructor is not required to see every submission by every student, they can easily lose touch with the abilities of their students. Additional tools not reported here have been implemented to allow the teachers to monitor the progress of their students and identify those who are falling behind on a daily basis.

Because student work is only reviewed by WebBot, there are stylistic issues that are not addressed, such as commenting and indenting. Additionally, students can sometimes short-circuit an assignment by writing a single routine to achieve a goal when the assignment was to create and use certain subroutines or data structures, or do something in a particular way. For these types of issues, a human may need to review the code.

Students have reported having a love-hate relationship with GradeBot. Most students love the fact that they get immediate feed-

back, and can know that their assignment is completed and accepted for full credit. A few students hate the fact that GradeBot requires extreme attention to such details as spelling and spacing in their output, and that occasionally the appearance of blank lines in the output can be hard to plan (e.g., should the blank line print outside the top of the loop, inside the top of the loop, inside the bottom of the loop, or outside the bottom of the loop).

The quality of student programming skills seems to have improved, but this improvement must be regarded as anecdotal. The fact that the students who complete the introductory classes have generally become capable programmers supports the hypothesis that automatic grading is feasible. However, we continue to monitor the performance of students in intermediate and advanced programming classes.

Table I: WebBot Submission History

Year	2004			2005		
Crs	CS 1	CS 2	Alg. Anal	CS 1	CS 2	Algor Anal
Students	175	133	19	169	117	42
Submits	n.a.	n.a.	n.a.	73405	15043	5684
Success	6995	1007	189	6518	767	385
Succ %				8.9%	5.1%	6.8%

One additional benefit is the creation of last mile learning. One difficulty in evaluating program assignments occurs when the assignment is only partially correct. Determining the amount of partial credit can be very subjective. With WebBot, there is no partial credit. As students get essentially unlimited attempts and immediate feedback, we require a program to be completely correct for credit. By debugging their own programs, students engaged in this "last mile learning." This is the learning that occurs when one finally finishes something and doesn't stop with close enough. One could easily argue that this is a behavior needed in the workforce. No employer wants a program that almost works. It is critical for students to learn to stick with the assignment until it is accurate and complete.

WebBot provides two other potential uses, not yet attempted. WebBot may be successfully deployed in distance education. It may be possible to automate an introductory programming course to such an extent that lectures could be recorded on video and the entire course could be delivered, conducted, and graded anywhere Internet access is available. Minimal human intervention would be necessary.

WebBot also allows for open entry / early exit. By using the idea of distance education, it should be possible for on-campus students to also start and complete the course on a schedule outside of the typical semester. Tutors available on campus could handle questions and an instructor would be needed only to resolve problems. Under this model, it would be possible to let students enroll at any time and complete at any time. Assignment deadlines could be tailored to each student's personal timeline. The GradeBot core provides evaluation of one assignment for one student at a time. Pacing, credit, cheat detection, and other functions are done in a management layer distinct from the core.

Development work continues on WebBot. Updates to the underlying tool and the interface described here are being performed. The most important development product underway is an improved instructor interface. Early versions required the instructor to be a programmer / hacker, and the current version still requires such a person to provide maintenance between semesters and to solve special situations that arise. The software engineering class is planning to develop an improved graphical interface to GradeBot. Both the student and instructor interfaces may be reported in the future.

We are sometimes asked whether we have any plans to share GradeBot and its associated tools. The answer is yes. We hesitate at the current time because it does not have a simple interface for maintenance activities, such as creating a new class at the start of a semester. The whole project carries the flavor of an extended proof of concept demonstration. It is efficient and robust for students, and we get along very well with it, but it would take a very interested colleague to install it and make it work somewhere else in its current form. However, we do want to make it faculty friendly and sys admin friendly,

which in itself is probably a substantial project, possibly open source.

REFERENCES

Chang, Carl, Peter J. Denning, James H. Cross II, Gerald Engel, Robert Sloan, Doris Carver, Richard Eckhouse, Willis King, Francis Lau, Susan Mengel, Pradip Srimani, Eric Roberts, Russell Shackelford, Richard Austing, C. Fay Cover, Gordon Davies, Andrew McGettrick, G. Michael Schneider, Ursula Wolz (2001). "Computing Curricula 2001 Computer Science," Final Report, 15 Dec 2001. Jointly published by IEEE-CS and ACM.

Colton, Don, Leslie Fife, Randy Winters (2005), Building a Computer Program Grader, Information Systems Education Journal **3**(6).

Forsythe, George E. Forsythe and Niklaus Wirth, (1965) "*Automatic Grading Programs*," CACM **8**(5), May 1965, pp. 275-278.

Harris, J. Archer, Elizabeth S. Adams and Nancy L. Harris, (2004) "*Making Program Grading Easier (But Not Totally Automatic)*," Proc. CCSC: Rocky Mountain Conference 2004, pp. 248-261.

Hext, J.B. and J.W. Winings, (1969) "*An Automatic Grading Scheme for Simple Programming Exercises*," CACM **12**(5), pp. 272-275.

Hollingsworth, J. (1960) "*Automatic Graders for Programming Classes*," CACM **3**(10), Oct. 1960, pp. 528-529.

Libes, Don. (1995) Exploring Expect. O'Reilly. ISBN: 1-56592-090-2.

Pendell-Jones, Allison. (2003) Academic Integrity and the Graduate Record Exam, in Ethics Today Online **1**(11), July 13, 2003, http://www.ethics.org/resources/article_detail.cfm?ID=826, accessed 2006-09-27.

von Matt, Urs, (1994) "*Kassandra The Automatic Grading System*," SIGCUE Outlook **22**(1), January 1994, pp. 26-40.