AUTOMATIC GRADING OF STUDENT'S PROGRAMMING ASSIGNMENTS: AN INTERACTIVE PROCESS AND SUITE OF PROGRAMS

Derek S. Morris[1]

Abstract *- A system for automatic grading of programming assignments is described here. This grading system consists of a suite of Perl and Java programs, linked by a database, and driven by an interactive, user-grader controlled grading process. The rationale for the process, the process itself - and the process's interaction with the programs, the database, and the user himself are discussed.*

This grading system has the ability to discover and accommodate un-anticipated solutions by means of a grading process that is highly interactive with its user – the grader. It also has the ability to automatically accommodate a wide range of simple student errors, which can easily befuddle a more naive grading system.

This system has been used since the Spring 2000 semester at Rutgers University, as a core part of the students' performance evaluation activities, in the Introduction to Computer Science course, where it routinely grades some 300 - 600 weekly assignments.

Index Terms - Automatic Homework Grading, Programming Assignments, Java

INTRODUCTION

This paper reports on the Homework Generation and Grading project (HoGG) at Rutgers University, that was first described as a FIE 2002 "work in progress" [1]. The ultimate goal of this project is to be able to easily create, deliver and rapidly evaluate student programming assignments, in a consistent manner with a low error rate, while still allowing students quite wide latitude in how they design their programs.

Our motivation in building the HoGG system was to supply a large number of students (600 per semester) with about 10 original programming assignments, during the course of each semester. Previously, we followed the path of tradition, having a group of Teaching Assistants *manually* grade the source code and output of student programs. It proved to be quite burdensome on the teaching assistant staff, and too often resulted in inconsistent, somewhat subjective grading of the individual submissions, and also in the untimely reporting of results. We looked towards an automated system that would allow the students to electronically hand in their source code, that would compile and execute the student program, soundly and fairly grade their homework, and promptly report the results back to the students.

Various versions of this system have been used at Rutgers University in our "Introduction to Computer Science" Course since the spring of 2001; it has graded some 20,000 homework submissions to date.

In this paper we will discuss the capabilities of the current system, the factors that have driven its design, and other important issues that we discovered through its application. Although the current system is meant for Java programming assignments, this paper describes principles that can be applied to assignments written in any of the currently widely used programming languages.

This paper is organized as follows: First we describe the process of developing an assignment, the assignment-specific portions of the HoGG system and use of the HoGG in the grading of the student submissions. We then discuss the types of simple student errors that could disturb the accuracy of the grading process. Finally we describe the very structure of HoGG and consider how it actually has performed in our experience.

THE HoGG PROCESS : DEVELOPING AND GRADING STUDENT ASSIGNMENTS

We will describe the process of developing and grading of student assignments, exemplified by a typical student assignment, where the student is to design and build a singly linked list class, where the insertion and deletion of nodes occur at opposite ends of the list.

The whole HoGG process starts with devising a didactic problem (dp), that is meant to illustrate some previously introduced concepts. The dp is fully described by a document, called the assignment specification. In our example assignment, suppose the idea was a singly linked list with the capability of insertion and deletion of nodes at either end of the list. Suppose we want the student to experience the relative ease of deleting a node from the front of the list as opposed to deleting a node from the rear of the list. We could ask the student to build a Queue using a singly linked list as the underlying structure.

[1] Division of Computer and Information Science, Rutgers, The State University of New Jersey, 110 Frelinghuysen Road, Piscataway, NJ 08854, USA morrisd@cs.rutgers.

The second step in the process is to develop a canonical homework solution, that will be used as a initial reference in the evaluation of the student submissions. In our example, this canonical program would have one or more constructors, the enqueue and dequeue operations and a toString()[2] method[3], that would provide a string representation of the contents of the queue. The string returned by the student's toString method will be used to provide visibility into the state of the student's queue. The format of this string must be simply and precisely defined by the specification. Since the dequeue method must deal with the possibility of an empty queue, we would need to consider the use of either an exception or an error return value. We might want to consider specifying an isEmpty() method or a size() method as well.

The third step is to develop a test program that is based upon testing the canonical program. In Java this test program would take the form of a separate class, containing a main method. This test program will be used to form the basis of the driver and evaluator in the HoGG system for this assignment. This test program should create a queue via each of the constructors, print out the initially constructed queues, insert a few nodes, delete a few nodes, and then attempt to delete a node from an empty queue.

The fourth step is to write the assignment specification, describing the signature of each of the required methods, e.g.,

```
Object dequeue()throws EmptyException
```

together with a natural language description of the required semantics, e.g., "If the queue is empty the dequeue method throws the Empty exception, otherwise the dequeue method removes the node from the opposite end of the queue, at which nodes are enqueued, and returns a reference to the information field of that node".

Again, because of its importance in the evaluation of the student submissions, the description of the string returned by toString must be simple and precise. As an option, we could provide the test program in a rearranged form, to help the students develop their program. This has a side benefit of making sure, that the students follow the naming and type specifications of each of the methods.

In the case of the queue, a new instance of the queue is created for each test, to avoid the possibility that a previous test could present the next test with a corrupted queue.

At this point, Teaching Assistants (TAs) are given the assignment specification and ask them to develop their own solutions. If there is not a clear consensus among the TAs as to what should be done to fulfill the assignment, we modify the specification. Once we see that the TAs find the assignment specification to be unambiguous, it is given to the students. The TA solutions will be used later in this process to develop the evaluator portion of HoGG for this assignment.

The fifth step is to convert the test program into a test driver[4] that will conduct a series of tests on the individual student programs by supplying method calls together with appropriate parameters. We distinguish the student program's response to each test from that of the other tests by having the driver insert markers into student program's output to identify these individual responses.

The sixth step is to design the evaluator. This design process is highly interactive and involves several steps. The design begins by driving the canonical program with the test driver and collecting its output. This output serves as the basis for evaluation. This basis is then "widened" by adjusting the evaluator to accept the expected minor errors[5]. We then further adjust the evaluator to accept the valid TA solutions.

The seventh step is to further develop the evaluator via a sampling of the student submissions. The output of each of these student programs that does not score well are manually evaluated against the specification. In those cases where the student program receives a less-than-perfect score but actually complies with the specification, the evaluator is adjusted to overlook the error.

The eighth step is to evaluate all the student submissions with the evaluator at its current stage of development. The database is used to compute the average score, where 10 is the highest score that is possible. Historically, in our teaching experience, that average should be 7 or higher. If it is less, a sampling is made of the imperfectly scoring student programs and step seven of this process repeated with those programs. Once the evaluator has been adjusted to accept those programs that comply with the specification, this current step is repeated until the average score of all the students is as expected.

The ninth step is to generate student reports from the database and email the scores to the students.

The tenth step is to collect ex post facto feedback on the quality of the grading for this particular assignment. This might take the form of a student complaining, that, while her program worked and produced the right output, it was graded with a less than perfect score. In this part of the process, the students present their cases to their teaching assistants, each of which have about 60 students[6] in their sections. The teaching assistant performs an independent assessment of the student program. Should the teaching assistant's assessment prove to be more valid than the

[2] In Java, the toString() method is automatically called to format text output for user defined objects.

[3] Although we will use the Java term "method" throughout this paper, we mean it in a generic sense. This term would be a "member function" in C++, a "function" in C, or a "procedure" in Pascal or Lisp.

[4] We will shortly see that this conversion involves translating the Java code into code that uses the Java Refection classes.

[5] We will see that this "widening" will be done with regular expressions.

[6] Using 15% as a working figure, the teaching assistant would have to evaluate 9 student programs per assignment.
Of course, not all the student complaints are valid.

HoGG-generated score, then the teaching assistant's grade is used, instead of the automatically generated one.

Here is an example of an HoGG-unanticipated solution, that was found after a student complaint was received:

The assignment was to build a general list using linked list techniques, where the student was supplied with a predefined Node class, which was to be used as the class of the elements to be inserted in and deleted from the list. One student defined her own Node class as an inner class. The resulting outcome was, that all her methods that used the class Node as a parameter or returned value, were not recognized by our driver, which was looking for the method

```
    void insert(Node x)
```

and not looking for the method

```
    void insert(LinkedList$Node x)
```

which was provided by the student and found by the Java reflection classes.

Our measure of quality of the automatic grading for a given assignment, is the percentage of the *grade replacements*, that occur for that assignment.

SPECIFICATION AND TESTING

The major challenge to automated grading is to get more fine-grained results than a simple pass/ fail report. Our approach to this challenge is derived fom the Software Engineering discipline. The student receives their assignment in the form of a software specification [7] which lists a number of requirements. This specification is published as a web page. We develop a number of functional, black box tests [2] relative to these requirements at the same time that this specification is being developed. We translate each required function into a description for a Java method[7]. Each test deals with a single aspect of a specific assignment and delivers a pass/fail result on that aspect. For example if the assignment was to develop a linked list, an insertion method and a deletion method would be specified and each of these methods would be separately tested.

We try never to specify fields[8] or use the fields of a student's program. If we were to do so we would be taking away the student's design freedom. In addition when we want the student to be able to reason abstractly about her data types, we do not want to be specifying any part of her data.

The tests and the requirements must be developed *concurrently*, so that each requirement is indeed practical to test and that each test actually tests a particular requirement. We strive to make each requirement and its associated test independent from any other requirement-test pair so that a failure in one requirement-test of the assignment will not automatically result in a failure in another.

We report the results of the tests in the form of a vector of bits each element of which represents the success (1) or failure (0) of a particular test. The report includes a description of each of the tests and their position in the bit vector.

We do not attempt to grade any compile time errors. The students are expected to successfully compile their program before they hand it in. If it does not compile, they receive a zero grade for their assignment. Receiving a zero for the assignment may seem inappropriate in a system that is generating a graded evaluation. What we are achieving with this "must compile" requirement, is avoiding creating a demand for interpreting the output of the compiler, a difficult task. We have in effect shifted our error evaluation process away from compile time onto the runtime. We achieve the ability to grade by explicitly requiring the student to develop her program incrementally, method-by-method, beginning with the simplest method. The student is required to maintain a compilable version of his program at all times during its development. Thus if a submitted program is not complete, it will be incomplete because it is missing one or more methods. However, the program will be whole in the sense that it compiles with the consequence that those methods that are actually present will be able to be evaluated by the grading program. Hence the student will be able to earn partial credit.

FAIRNESS IN GRADING – DEALING WITH "MINOR" ERROR PATTERNS

There is a number of errors that a student could make, which can cause an automatic grading program to misgrade the actual value of the student's effort. These errors, such as misspelling the name[9] of a method, may not be an indication of the skills, for which the student is being evaluated. One could (and students do) say, that such a grading result is unfair. There are several mistakes to which beginning students are prone, that, although minor, can cause any automatic grading program to be more severe than intended. They can be categorized as follows:

Naming errors: They occur as Java is a case-sensitive language. For example, we observed students who seem to "think auditoraly" rather than "visually", who name a required class or method "thingone" instead of a specified "ThingOne", thus introducing an error into their program. Also spelling mistakes will result in errors in their program.

Output format errors: One must specify the format of the student program output, so that the results of the program can be evaluated. Because the program driver will produce marker sequences in the output, one must be careful to

[7] Again the Java term "method" is meant in a generic sense.
[8] Again this is Java terminology for a generic idea. A Java Field is a variable in C, a member in C++.

[9] Since the student meets a requirement by writing a method, we need to be able to identify the method in order to be able to associate it with its intended requirement. The most common form of method identification is via the method's name.

specify the program output and the markers, so that they are mutually exclusive. Beginner students typically are insensitive to the different types of white space (the number of spaces, combinations of spaces, tabs and newlines). They also tend to confuse the various bracketing mechanisms ([],{},<>,()), and when there is no compiler support, as is the case with an outputted string, they may not always balance them. The student will sometimes pepper their output with extraneous words and symbols. A grading program can misinterpret these as marker sequences that were placed by the testing driver to identify certain outputs rather than as student output. If this misidentification takes place, a large portion of the grading result may be in error.

Indirect Dependency: This is a situation where the correct performance of one method is dependent on the correctness of another method. For example, if the student assignment were to build a linked list, then one cannot test the delete method without a correctly functioning insert method, to give the delete method something to remove from the list.[10]

Direct Dependency: This is a situation when a student has one method call another method. Should the called method fail and if the calling method is error free, we do not want to also evaluate the student's calling method as failed.

Infinite loops: When a method enters a loop, there is the possibility that the method will never return and the grading process effectively stops progressing. The grading program needs to detect infinite looping and to be able to isolate it to a single student method and recover from it.

Infinite Recursion: When the student is working with recursive methods, there is the possibility of a system stack overflow[11].

GRADING IN THE PRESENCE OF MINOR ERRORS

In spite of minor but plentiful errors ever present, HoGG strives to perform meritorious and accurate grading, through the use of the following technologies.

The first technology is the Java Reflection Classes [6],[3], which can be used to execute methods based on their signature (the types of their parameters, returned value, the exceptions that they throw, and whether they are static or non-static), irrespective of the method's name. If one specifies these methods carefully, then they all may have a unique signature, and therefore can be identified regardless of the actual name, given by the student. If, due to the nature of the problem, a unique method identification is not possible based on signature alone, then the String methods in Java, together with the refection mechanism, can be used to ignore the case of the name[12]. We also use a timer from the Java library, in conjunction with the reflection mechanism, to "time out" a possibly infinite loop or recursion[13] by timing each method invocation.

If one were dealing with student programs written in a language other than Java, the reflection technology would not be available. One would have to interrogate the student's source code using regular expressions, described below, to find textual patterns that correspond to the methods' signature in the program. The test program using the required names, parameters and exceptions would then be built on-the-fly.

The second technology is Regular Expressions, which is primarily used to evaluate the student program output. In addition to dealing with the student program's output, Regular Expressions are used in conjunction with the reflection classes, to provide tolerance of common name misspellings. We have been using Perl (Pattern Extraction and Reporting Language)[4], as well as the newly introduced Regular Expressions included in Java 1.4 to provide this technology. For one of the best references on using this technology see[5] .

The third technology is inheritance and polymorphism, which is used to define and override a directly or indirectly dependent method, by a known, correct equivalent method. This can only work, if the fields on which the dependent method operates, have been specified in the assignment specification. These fields, in a quality object-oriented design, would be declared as private fields, and would then have to be changed into protected fields, by modifying the student's source code, a simple task when using Perl code. For example, consider the linked list with its insert and delete methods, where the delete method is dependent on the correctness of the insert method. We could override the student's insert method only if we understood the student's design concept. As we said in the section "Specification and Testing", we are reluctant to specify the fields, as that would restrict the students' latitude in their designs. To practice this reluctance, we actively avoid making a requirement that results in dependent methods. In the case of the linked list–insert–delete example, we have taken the attitude, that the student must have an insert method working, before they can verify the correctness of their delete method. We go so far as to suggest, that they use a specific design-code-test sequence for each assignment, where the more dependent methods occur later in this suggested sequence.

[10] It can be argued that the student will have had to have first made a correctly functioning insert in order to test out their delete for the same reason. However suppose the delete would have worked if the insert had worked . Should the student then be penalized for both a bad insert and a bad delete?

[11] In Java this is an unrecoverable error which causes the Java Virtual Machine that is running the student program to be terminated.

[12] With Java 1.4, one can specify a set of name variations via regular expressions.

[13] It is easier with loops than with recursion because setting the time out value for a recursively called method needs to be more precise for the recursive case in order to avoid stack overflow.

THE STRUCTURE OF THE GRADING PROGRAM

The overall design of HoGG consists of the following components:

The Framework

The framework orchestrates the other components and communicates with them via a MySQL database.
Students hand in their assignments electronically, via a computer dedicated to serving undergraduate courses. The framework will first build a data base table for each student assignment with a row per student, which contains the student's source code. The framework then invokes the Java compiler (javac), which compiles the source from the database and places the resulting byte code and any error text into the database table. If the compiling is successful, the framework runs the driver, which loads the byte code and invokes the student program's methods in accordance with the test plan, while placing the results of the run into the database table. Finally the framework runs the evaluator, which evaluates the output of the run and places the evaluation results into the database.

Perl was chosen as the framework implementation language, because of its facility in navigating file systems, interfacing with databases, managing the execution of other programs and for its cross platform portability.

The Driver

The driver is a Java program which provides the inputs to the student program. The driver loads the student's class file into the same Java Virtual Machine (JVM) that is running itself, and then invokes the student program's methods, one at a time, where the execution of each is limited to a certain time interval. The framework captures the student program's output as the various methods are exercised, injects markers in the output stream, so that the output parts can be later identified by the evaluator, and stores the result in the database.

The Evaluator

The evaluator rates the student program's output against the "widened" canonical output. The framework then invokes the evaluator, which acts upon the student program output that was stored in the database. The evaluator, because of its need for regular expressions[14], is a Perl subroutine. The assignment-specific portions of the evaluator consists of a series of tests, each of which use one or more regular expressions to examine the student program output. These tests are designed to accommodate the canonical output, common student errors and legitimate variations learned from the teaching assistant solutions. The evaluator constructs a bit vector, one bit for each of the tests, where a 1 indicates a passage and 0 indicates a failure.

Consider the testing of a linked list as our example. A testing strategy could be to insert a few nodes into the linked list, while looking at the state of the linked list by calling the toString() method after each insertion. There would be a bit in the bit vector for each toString() call, that would represent the correct functioning of the insert operation and the correct functioning of the toString method as well. We could then call the delete method followed by a call on the toString method to check the correct functioning of the delete method.

A score on a scale of 1 to 10 possible points is computed from the bit vector, where each test is weighted equally.

The DataBase

We chose a database to store the intermediate and final results, to give us reporting flexibility. With a database, it is easy to view various aspects of these results, e.g. averages, maxima, scores by section, and the identification of students with the identical bit vector pattern (a possible indication of cheating). In addition, once a database is in place, both the evaluator and reporting code becomes more system, and platform independent, as they are freed from a rigid file structure. HoGG uses the popular MySQL [8] database, chosen for its reliability, flexible Perl interface, low cost (it's free) and for the benefits of the open source philosophy that created it.

The Reporter

Once the results are put into the database, we have several *reporting* options. We choose to write an explanation, that describes the test associated with each of the vector's bits, then we email to each of the students his explanation together with his vector and score.

We also generate a report, containing the student's name, score and vector for each of the class sections, and then we email that report to the student's teaching assistant.

HoGG PERFORMANCE

The HoGG is used regularly to evaluate about 300 (last year 600) student submissions per assignment, 12 assignments per semester, for a total of 3600 assignments. As we gain more experience applying the HoGG system, we find that we are able to devise a homework specification, canonical solution, driver and evaluator, all this with about 8 person hours of effort[15] per assignment. The grading result yielded will be accepted by more than 95% of the students. This is a recent improvement over our previously reported results [1]. It has been achieved by taking more care with the specification, and then through being able to demonstrate to

[14] This now could be written in Java as the latest version (1.4) has regular expressions.

[15] We often achieve the same result with as little as 4 person hours.

the question- or objection- raising student the exact output of his submission, as it relates to the specification, via the database.

In our experience HoGG is able to grade a run of 300 student submissions in about a few minutes of time on a 1.4 GHz personal computer, running Windows XP.

CONCLUSIONS

It is practical and extremely helpful, it is human effort- and computer time-saving, to use a well-designed, automated mechanism to grade large numbers of student programming assignments, where we achieve a consistent, objective and valid grading result. It requires, however, that the specification for the assignment be written simultaneously with the test plan, and also requires that typical minor errors be *anticipated*.

HoGG has effectively graded over 20,000 student programming submissions while allowing the students wide creative latitude in their programming solutions. For each assignment, HoGG requires about 4 - 8 person hours to develop the assignment and assignment specific portions, to achieve an error rate of 5%.

FUTURE WORK

We are currently working on having cheating-detection software fully integrated into the HoGG system. Source code duplicate detection is an N-squared problem, and hence not practical for a large number of submissions. However, as we have mentioned before, identical, non-perfect bit vectors are a possible indication of student program duplication. Since the database contains both, the vectors and the source code, it would be operationally simple, to automatically check the submissions with identical vectors, for mutual similarity.

REFERENCES

[1] Derek S. Morris, "Automatically Grading Java Programming Assignments via Reflection, Inheritance, and Regular Expressions", proc. FIE 2002, Nov. 2002.

[2] Boris Beizer, "Black Box Testing: Techniques for Functional Testing of Software and Systems". Wiley, 1995

[3] Patrick Chan, Rosanna Lee, Douglas Kramer, "The Java Class Libraries Second Edition", Volume 1, pp.19, Addison Wesley, 1998.

[4] Larry Wall, Tom Christiansen, Randal L. Schwartz, "Programming Perl", 2nd ed., O'Reilly & Associates, 1996.

[5] Jeffrey E. F. Friedl, "Mastering Regular Expressions", O'Reilly & Associates, 1997.

[6] David Flanagan," Java in a Nutshell", 4th ed., pp, 158, O'Reilly & Associates, 2002.

[7] Dick Hamlet, Joe Maybee, "The Engineering of Software", pp.119, Addison Wesley, 2001

[8] Paul DuBois, "MySQL", NewRiders, 2000.