GRADING STUDENT PROGRAMS - A SOFTWARE TESTING APPROACH

Edward L. Jones
Florida A&M University
Tallahassee, FL 32303
850-599-3050
ejones@cis.famu.edu

ABSTRACT

This paper describes an experience of automating the grading of student programs. A testing framework provides guidance for developing the assignment specification and the grading program. Automation saves time and improves grading consistency and feedback to students. After an adjustment period, student programs improved. Although the instructor invests more time writing a testable assignment specification and developing the grading program, these costs are expected to be amortized over multiple courses and assignments.

1. INTRODUCTION

The problem of managing large numbers of students in programming classes is a common one. Most course grading environments include document flow, grading, and record keeping. Document flow includes distribution of assignments, program submission, and the return of graded programs with feedback. Some environments provide a simplified, user-friendly interface to facilitate the programming process for novices [1]; others use more advanced programming environments such as Unix [2]. Some environments use human graders [3], while others fully automate the grading process [2, 4, 5]. The student may receive immediate feedback on programs at the time of submission, as with the TRY system [2] and the homework checker of Arnow [4]. Although these systems disclose only minimal information, the student is alerted to problems that prevent the program from compiling, running, or passing a test case, and is allowed to fix the problems and resubmit the assignment.

The purpose of this paper is to illustrate how concepts of testing can be used to specify programming assignments that are amenable to automated grading. The SPRAE framework [6] is used to guide the preparation of a *testable* specification and the derivation of program checking scripts, as in [2, 4]. These processes are incorporated into an existing program submission and grading environment (PSGE) developed by the author. PSGE, which closely resembles the TRY system [2], will be described in the next section, but it is not the major focus of this paper.

2. THE PROGRAM SUBMISSION AND GRADING ENVIRONMENT

PSGE is a Unix-based program submission and grading environment supporting semi-automated and fully automated program grading. Assignments are posted in a

public repository accessible to students. Each student is provided a private submission repository for submitting assignments and receiving feedback. The student's environment contains a set of Unix commands for accessing the repositories. PSGE incorporates revision control, permitting a student unlimited submissions, subject to deadline controls. A time-stamped log of all submissions is maintained.

Students are free to develop programs in the environment of their choice, but are responsible for making sure the submitted program compiles and executes in the Unix environment. Students are required to follow strict naming conventions for all source and data files. Failure to follow conventions can result in failed compilation or execution, and lead to a lower assignment score.

Grading is deferred until after the assignment deadline has been reached. In semi-automated mode, PSGE provides the instructor an on-line grading environment, but the instructor must inspect the source program (to determine the documentation score) and program outputs (to determine the correctness score). PSGE fetches each program from the submission repository, compiles it, and initiates program execution. The instructor is required to provide interactive inputs needed by the program. The instructor manually applies deduction schedules for compilation, documentation and correctness (adherence to the problem specification). Using the deductions entered by the instructor, PSGE assigns a program score, creates a program grade report containing the deduction for each category and an anecdotal explanation of the overall program quality. PSGE deposits the grade report into the student's submission repository. PSGE maintains a master log for all graded programs.

In fully automated mode, PSGE removes the instructor from the grading loop as inspector and input source. (The instructor or assistant is needed to cancel non-terminating programs. When grading is resumed, cancelled programs are removed from further consideration.) A standard deduction is made when the program does not compile or execute. PSGE invokes a *program grader* to execute the program. Interactive inputs are supplied from data files, and interactive outputs are captured in results files. Next, PSGE invokes a *program checker* to identify discrepancies in program outputs, and to reduce the program score accordingly. PSGE generates a grading log detailing the discrepancies and the resulting deductions, and deposits the log into the student's submission repository.

3. SPRAE - THE TESTER'S FRAMEWORK

Testing is a very broad subject area encompassing a variety of objectives, strategies and levels of detail [7]. The program grading problem is a special case of the testing problem, where the goal is to identify significant departures from the assignment specification, and to define a measure of deviation, the points deduction. We now present a set of essential testing principles that should be present in the practice of software testing. The acronym SPRAE has the expansion:

- **Specification**. *A specification is essential to testing.* The specification defines what "correct" means.
- **Premeditation.** *Testing requires premeditation*, i.e., a systematic test process.
- **Repeatability.** *The testing process and its results must be repeatable and independent of the tester,* i.e., consistent and unbiased.

- **Accountability.** *The test process must produce an audit trail*, i.e., evidence of what was planned, what was performed, what the results were, and how the results were interpreted.
- **Economy.** *Testing must not require excessive human or computer resources*, i.e., the test process must be cost effective.

4. A SPRAE-COMPLIANT ASSIGNMENT LIFECYCLE

We now use the SPRAE framework to define a programming assignment lifecycle for the instructor. The assignment lifecycle integrates elements of the testing lifecycle in order to ensure the testability of the assignment specification, and to direct the development of software to automate grading the programs. Figure 1 shows the integrated lifecycle. Note that test activities (in italics) must be performed concurrently with assignment development.

<table>
<tr><td>Test Lifecycle</td><td>Analyze</td><td>Design</td><td>Implement</td><td>Execute</td><td>Evaluate</td></tr>
<tr><td>Test Products</td><td>test plan</td><td>test data sets
test cases
test scripts
grading plan</td><td>test driver
program checker</td><td colspan="2">Program grading logs</td></tr>
<tr><td>Assignment Products</td><td colspan="2">Assignment specification</td><td>Assignment spec
Sample data files</td><td colspan="2">Program grade reports
Submitted programs</td></tr>
<tr><td>Assignment Lifecycle</td><td colspan="2">Prepare Assignment</td><td>Distribute Assignment</td><td colspan="2">Grade Assignment</td></tr>
</table>

Figure 1. SPRAE-Compliant Assignment Lifecycle

The next three sections provide a walkthrough of the assignment lifecycle for an actual student assignment, an interactive file update program (p7). The program is menu driven. Each transaction must be written to an audit file, flagged as valid or invalid. The program must validate each transaction, and attempt to apply valid transactions to the master file. When a valid update transaction can not be applied (e.g., opening an already existing account), the update must be flagged as a failure and written to the audit file.

5. STAGE 1 – PREPARE THE ASSIGNMENT SPECIFICATION

This stage lays the foundation for achieving automated grading. Considerations for automating program grading require that the assignment specification be developed concurrently with the design of the automated grader. The first subsection identifies essential elements of the assignment specification; the subsections after that identify essential test products that influence the formation of the assignment specification.

5.1 Specification

Three essential components of the assignment specification are input requirements, output requirements, and processing requirements. Because each component can be the focus of specific test cases, the specification needs to be unambiguous with respect to each component.

Input requirements typically specify data types and range limits for data items, the format of input data, and user interaction sequence. The p7 input requirement below specifies the order in which transaction data must be entered from the keyboard:

```
menu choice:      1
data sequence:    Account#, Amount, Lastname, FirstName
```

Output requirements typically specify the destination of interactive and file outputs, and the format for all outputs. The specification must be explicit regarding text case, wording, spacing, etc. Output requirements for p7 include the exact wording of error messages, e.g., "ACCOUNT ALREADY EXISTS" and "INSUFFICIENT FUNDS"

Processing requirements specify the conditions under which the program must transform inputs into outputs. Processing requirements contain explicit and implicit statements of stimulus-response situations the program must handle. Stimulus-response pairs implied by the specification determine test cases. Two p7 processing requirements follow:

```
(1) A withdrawal amount may not exceed the account Balance
(2) OPEN account must fail if the account already exists.
```

5.2 Test Analysis – What to Test

A *test plan* defines the strategy for testing a program. The simplest strategy is a *partitioning* strategy [7], where the testing effort is dividing into at least two test runs, nominal (perfect-world) testing, and error-handling testing. The p7 specification given in section 4 suggests partitioning testing into three test *runs*:

- nominal testing of valid transactions that should be applied successfully;
- update failures – valid transactions that can not be applied; and
- invalid transactions that must be rejected and no update attempted.

Table 1 gives an excerpt from the test plan for assignment p7. The test plan identifies the resources needed to test the assignment and the execution sequence of the test runs. Splitting the tests into runs simplifies checking correctness.

Table 1. Test Plan for Assignment p7

```
1. There will be three test runs

2. All test sets will use the same master file.

3. Test Execution & Results Verification

3.1 For each test set there will be a test driver script (testscript) that
    executes the program, supplying the test cases as program input.
3.2 For each test set there will be a checker script ( checkscript) that
    scans the program's output file (p7_audit.rpt) for expected results.
3.3 The checker script deducts points for failure to produce expected
    output.
3.4 The results from the checker script will be written to file
    p7.log-studentid.
```

5.3 Test Design – How to Test

The test design consists of test data sets, test cases, and test scripts. A *test data set* is a data file used by the program. A rule of thumb is to use a small, simple set of data that facilitates determining the stimulus-response correctness. *Test cases* are pairs, <stimulus, expected response>, implied by the assignment specification. The response produced by a test case may be a program output, or a side effect upon external or internal data. A test case is illustrated below.

```
Test Case #:      14
User Inputs:      5  7777777
Expected Results: VALID txn message, update FAILURE,
                  msg = ACCOUNT DOES NOT EXIST
```

A *test script* defines a sequence of test cases associated with a test run.

5.4 The Grading Plan

The *grading plan* identifies the subset of test cases in a test script that will be verified for the purposes of deducting points. (Some of the test cases correspond to steps that merely set the stage for a significant program action.) An item in the grading plan merely annotates a test case with the amount of the deduction if expected results are not achieved, as shown below:

```
Test Case #:      14
User Inputs:      5  7777777
Expected Results: VALID txn message, update FAILURE,
                  msg = ACCOUNT DOES NOT EXIST
Deduction:        1
```

6. STAGE 2 – DISTRIBUTE THE ASSIGNMENT

6.1 Document Flow

The assignment is distributed when the assignment specification and sample data files are placed in the PSGS public repository, and students are notified by electronic mail that the assignment. A submission deadline is established. Students may submit their programs as often as they desire, subject to the deadline (lates are accepted, but penalized).

6.2 Grader Implementation

The grader consists of a test driver and a program checker. The *test driver* executes the program against multiple data sets or user inputs. For each such test run, the driver must: (1) set up the environment for running the student program; (2) run the program, supplying interactive inputs, and capturing program outputs to results file; (3) invoke the program checker; (4) accumulate points deductions; and (5) produce the program grading log.

Each test run has a program checker script that uses Unix pattern matching to find expected results in the output files. For test case #14 shown earlier, the expected output looks like `"… 014 … FAILURE … ACCOUNT DOES NOT EXIST …".`

The checker scans for the pattern `"*014*FAILURE*ACCOUNT*NOT*EXIST*"`, and deducts a point if the pattern is not matched.

7. STAGE 3 – GRADE THE PROGRAMS

During this stage the grader is executed under the control of PSGS. The grader invokes program *checker scripts* that check for results based on the grading plan. The outcome of each check is logged in the grading log, providing a detailed explanation for the student. Mismatches between expected and actual results are labeled with the Unix login of the student, followed by the word **WRONG**, followed by an explanation of the deficiency. This detailed log documents the context of each result verified by the checker, whether correct or incorrect. The human grader is relieved of the task of providing detailed grading notes. Excerpts from a grading log are shown in Table 2.

Table 2. Grading Log

```
p7 Grading Log for danaej
_________________________

0007  SUCCESS                                        ACCOUNT OPENED
danaej – WRONG – missing CLOSED MESSAGE
0008  SUCCESS                                        BALANCE DISPLAYED
0004  FAILURE                                        INSUFFICIENT FUNDS
danaej – WRONG – extraneous DEPOSIT MESSAGE
0001  UPDATED    7777777     $5.00  030300          OPENDEPOSIT  TOORLOW
danaej – script3 PENALTY POINTS = 5
danaej – TOTAL RUN PENALITY POINTS = 9
```

8. OBSERVATIONS AND CONCLUSIONS

The process described in this paper satisfies the principles of the SPRAE framework. It is specification driven. It involves a systematic concurrent process of specification refinement and test design. The automated grader ensures repeatability and consistency. Permanent documentation is generated for each program. Automation saves the instructor time and effort.

Initial experience with the automated grader is mixed. The first reaction from students is shock. Because the grader is totally objective, students are appalled when they are penalized for "minor offenses" that have been overlooked in the past. To be fair, the instructor has the moral imperative to write a solid, testable specification. To do so is just as difficult for the instructor as following the specification is for the students. A commitment to automated grading stretches faculty and students alike.

After a few programs students show the ability to be more careful and attentive about the assignment specification, despite their initial resistance. Students appreciate the detailed grading log that explains each deduction. The instructor learns to write better specifications, often by replicating the structure of previous assignments, enabling the reuse of the program grader design. Although the instructor must invest more up-front work, the quality of assignments and the quality of student work appear to benefit from automated grading.

The primary lesson learned is this. A commitment to automated grading necessitates a commitment to developing better-than-average assignment

specifications. Since the specification must be completed before the assignment is distributed, the assignment specification process must be performed concurrently with a specification-based test design process.

9. FUTURE WORK

Three areas of future work will be pursued. First, the use of automated grading will be expanded: it will be used for a second time in one course, and for the first time in another. Second, the grader program will be made available to students sooner, at the time they submit programs, as a *self-service* grader. (The instructor may use a different test data set in the actual grading.) Providing this service will remove the surprise element from the student, and should reduce appeals and challenges of program grades. Finally, in order to facilitate the adoption of this approach by others in the department, we will investigate automating the generation of the grader from the problem specification.

REFERENCES

[1] Churcher, N.I., Cockburn, A.J.G., McMaster, B.N., Horlor, J., "CUTE: The Design and Evolution of a First Year Programming Environment," *Proceedings 1998 International Conference on Software Engineering: Education & Practice*, Dunedin, New Zealand, January 26-29, 1998.

[2] Reek, K.A., "The TRY System – or – How to Avoid Testing Student Programs," *Proceedings SIGCSE Bulletin vol. 21*, no. 1, February 1989, pp. 112-116.

[3] Canup, M.J. and Shackelford, R.L., "Using Software to Solve Problems in Large Computing Courses," *Proceedings SIGCSE '98*, Atlanta, GA USA, pp. 135-139.

[4] Arnow, D.M., ":-) When You Grade That: Using E-Mail and the Newtwork in Programming Courses," *Proceedings 1995 ACM Symposium on Applied Computing*, February 26-28, 1995, Nashville, TN USA, pp. 10-13.

[5] Jackson, D. and Usher, M., "Grading Student Programs Using ASSYST," *Proceedings SIGCSE '97, 1997, pp. 335-339.*

[6] Jones, E.L., "The SPRAE Framework for Teaching Software Testing in the Undergraduate Curriculum," *Proceedings ADMI 2000*, June 1-4, 2000, Hampton, VA USA,.

[7] Zhu, H., Hall, P.A.V., and May, J.H.R., "Software unit test coverage and adequacy," *ACM Computing Surveys, Vol. 29*, No. 4, December 1997, pp. 366-427.