A Gimmick to Integrate Software Testing Throughout the Curriculum

Michael H. Goldwasser
Dept. of Computer Science
Loyola University Chicago
6525 N. Sheridan Rd.
Chicago, IL 60626
mhg@cs.luc.edu

Abstract

We discuss our experiences in which students of a programming course were asked to submit both an implementation as well as a test set. A portion of a student's grade was then devoted both to the validity of a student's program on others' test sets, as well as how that student's test set performed in uncovering flaws in others' programs. The advantages are many, as this introduces implicit principles of software testing together with a bit of fun competition. The major complication is that such an all-pairs execution of tests grows quadratically with the number of participants, necessitating a fully automated scoring system.

1 Introduction

We wish to share our experiences involving a fun way to incorporate aspects of software testing throughout the programming portion of the curriculum. Our simple exercise is to have each student submit both source code as well as a test set. An experiment is then performed where each submitted program is run on each submitted test set. A portion of a student's grade is then devoted both to how well that student's source code performs on others' test sets as well as how that student's test set performs in uncovering flaws in others' programs.

Such an exercise can be used to either implicitly or explicity demonstrate many formal issues of software testing, no matter how much (or little) time is devoted to such issues through lecture. At its heart, the experience involves black-box system testing. A student must develop a test set, in advance, which will be run on other students' submissions. Of course, since directions for many programming assignments are quite leading, a student may have some knowledge of the underlying structure of others' programs. In this case the exercise is more in line with "gray-box" testing. Furthermore, by designing a testing interface that allows method-by-method calls, unit testing can be accomplished.

Though perhaps a simple gimmick, this technique is appropriate at all levels of the curriculum and can be applied to most existing programming assignments with a bit of care. The experience appears to offer numerous advantages including, but not limited to,

- The competitive scoring provides a bit of lighthearted motivation to course work (similar to uses of tournament play in which students' implementations compete against each other [16]).
- Even students who are struggling with their own implementations can feel fully included in developing their own test sets.
- A collection of students' implementations offers a wonderfully diverse environment for software testing.
- The scoring system provides a *quantitative* evaluation of both program validity and test set quality that can be included as part of the overall grade.
- The submitted test sets may uncover bugs in some implementations, which went unnoticed during the instructor or TA evaluation.

Adversely, we find only one disadvantage: scoring this experience is *quadratic* in the size of the class! For all but the smallest of groups, this process must be fully automated to be feasible. Once this automation is provided, however, the technique scales to classes with enrollments anywhere from 5 to 500 students.

Our presentation proceeds as follows. In the coming section, we reflect on the comments of previous authors regarding software engineering and program testing in the computer science curriculum. In Section 3, we discuss the logistics of such an exercise and show how it can

SIGCSE'02, February 27- March 3, 2002, Covington, Kentucky, USA.

be adapted quite flexibly to most existing programming courses. For ease of exposition, Section 4 provides two concrete examples of programming assignments drawn from our own experience with this technique over two semesters. Section 5 contains a few preliminary observations from our experience. Finally, we discuss in Section 6 some important issues in developing a fully automated system for scoring such experiments.

2 Previous Work

The most recent draft of *Computing Curriculum 2001* lists Software Validation (SE6) as one of the core units of the computer science curriculum [1]. Yet, this unit is given very little coverage in introductory courses, often with only an expectation of ad hoc debugging of one's own implementation. The subject is traditionally approached as part of a Software Engineering class later in the curriculum.

Several voices have argued that software engineering principles, including testing, should be integrated throughout the curriculum [5, 6, 9, 15]. Two previous SIGCSE panels have focused on this issue [13, 14], about which Nell Dale writes, "testing must begin in CS1 and be reinforced in each succeeding course." A variety of authors have discussed approaches to incorporating testing into the curriculum, including:

- Students are required to submit formal test plans or test logs [3, 5, 9, 11, 18]. Presumably, an instructor or TA must grade these subjectively.
- A student's (white-box) test plan for her own code can be graded quantitatively by measuring the percentage of code reached by the tests [3, 8].
- Students can perform (black-box) testing of a flawed executable provided by the instructor [9, 11, 15].
- A student can test one other student's program [11, 18]. This provides the testing student with good experience, but it is not clear how to fairly evaluate the tester's performance in a way that is independent of the selected implementation.
- Upperclassmen can be utilized as part of a "Test Lab" developing testing plans for lower level class assignments [11].
- Instructors can provide automated testers, giving feedback to students during program development [2, 12, 17].

Our technique can be easily integrated into courses which emphasize additional principles of software engineering. Alternatively our experience can be incorporated independently as a so-called "frosting" to an existing course [3].

3 Assignment Logistics

Our general approach involves programming assignments for which each student submits both an implementation as well as a test set[1]. For a student, Alice, we compute the following two metrics:

1. The percentage of test sets on which Alice's code worked properly.
2. Of those implementations with detected flaws, the percentage fooled by Alice's test set.

A portion of Alice's grade can be based directly on these two metrics, with the remainder of the grade determined by standard instructor or TA evaluation of correctness, efficiency, style, documentation, etc. Bonus points can also be awarded for accomplishments such as exposing a flaw in a program that passed all other students' tests.

In theory, such an approach should be amenable with all existing programming assignments. In practice, assignments must be well specified so that the notion of "correct" behavior is unambiguous and so that automation is achievable. Though instructors may face a bit of a learning curve in preparing such assignments [7, 10, 17], we have found great flexibility in the framework.

In the remainder of this section, we discuss some of the key issues in assignment development.

3.1 Modular assignments

This experience can be applied to assignment settings in which students develop self-contained programs from scratch as well as settings in which they implement particular modules that are to be combined with modules provided by the instructor. For reasons detailed in the following sections, we strongly recommend the latter style of assignment in this context. This will allow for the instructor to provide front ends with a particular user interface and to incorporate non-trivial back end components to support automated scoring.

3.2 Test set format

Because students will be submitting test sets that are to be run (automatically) on other students' submissions, a standardized format for describing the tests must be established in the assignment description. The overwhelming conclusion is to rely on a textual interface for program input.

The input and subsequent parsing must rigorously follow prescribed standards. Potential pitfalls for a student include submitting test input that is essentially

[1] Likewise, this approach can be directly applied to a course setting in which students work in teams

ignored for not adhering to the standards, as well as submitting an implementation that does not properly parse the standard input format. In either such case, the student risks failing the competition en masse. In fact, there is a compound risk that the student will submit a flawed input format that is self-consistent with a flawed parser. Unless the intent of an assignment is specifically geared towards a student's ability to parse or error-check input, having the instructor provide all students with a robust front-end parser will avoid most such tragedies.

Such a traditional "console" interface may indeed be unappealing to modern students. We wish to point out that in a modular setting, the instructor can provide both a textual interface as well as a separate graphical user interface. Students could use the GUI for most of their development if they wish, but would eventually need to use the textual driver for developing their test file. As a side effect, it is advantageous for the student to see both such interfaces. Even though the GUI might seem attractive for the end user, the textual interface is quite valuable during a debugging phase on large tests. It is quite time consuming to repeatedly step through the first 40 mouse-clicks to reach the bug on the 41^{st}, yet easy if such a chain of commands can be read from a file.

3.3 Test set size

As the competition is structured, a student receives credit if her test set exposes one or more flaws in another student's implementation. Thus students will gain an advantage by developing larger and larger tests. For this reason, a strict limit should be placed on the inherent size of the tests.

3.4 "Output" format

A constraint of automated scoring is the need to recognize a program with proper behavior on a given input. We will discuss technical issues more in Section 6, however the most direct method is to pattern match textual output of the student's program to the output of a "model" implementation.

When students are responsible for producing output, strict standards must be established by the instructor and followed by the students. In a modular setting, many pitfalls can be avoided by again relying on the output of a standardized front end provided by the instructor. As was the case with the input, an instructor may choose to independently provide students with a graphical interface.

Checking the correctness may require additional, non-trivial tools for certain assignments. For example, some tasks may involve a notion of non-determinism in which case there exist multiple "correct" behaviors on a given instance. In such a context, there is no suitable notion of a "model" implementation for comparison. Instead, additional software may be used to monitor the behavior of a student's implementation to determine correctness. Such a system can either be incorporated as an additional module in the executable or as a post-processing step based on a suitable execution trace.

Programming assignments can be developed that do not involve any traditional output. For example, a student's task might be to develop a complex data structure based on given input. The correctness can be determined by an additional module that compares the internal structure produced by the student to a "reference" structure built by the checker [2]. Meaningful output can be produced in the case of a mismatch.

Whether or not to provide students in advance with such non-trivial correctness checkers is up to the instructor's discretion.

4 Two Assignment Examples

Merging two sorted linked-lists.

The students are asked to write a routine that accepts two linked lists, each containing a non-decreasing sequence of positive integers. The goal is to return a single, non-decreasing linked-list comprised of all of the nodes of the original lists. A provided driver repeatedly reads delimited sequences of non-decreasing integers. For each pair of sequences, the driver verifies the original non-decreasing order, constructs linked lists and passes them to the student routine. The test sets can contain an arbitrary number of sequence pairs, however the combined length of the lists is limited to 100 elements.

For checking correctness, there is no need for a model implementation. The driver can be augmented to examine the resulting linked list, verifying the structure and order of elements, and computing a checksum to compare to the original set of values.

Maintaining a hand of cards.

We borrow Exercise P-5.6 from the Goodrich and Tamassia text [4]. Students use a list ADT with positional fingers to represent a hand of playing cards, supporting the following two operations:

- `newCard(c,s)`: add the new card c of suit s to the hand.
- `playDown(s)`: remove a card of suit s from the player's hand; if there is no card of suit s, then remove an arbitrary card from the hand; if the hand is empty, produce an appropriate error.

A given driver provides an interface to calling each of these routines as well as resetting to an empty hand, and displays the card returned by `playDown`. The student tests are limited to 100 operations.

This assignment is an example that involves an aspect of non-determinism due to the choice of legal plays. Comparing to a model implementation is not a valid check of correctness. Rather, a new module is needed which monitors the behavior of the student's program while keeping its own record of the implicit hand. Errant behavior can be reported when detected.

5 Preliminary Experiences

We have used this gimmick in two offerings of our Data Structures (CS2) course at Loyola during the academic year 2000–2001, with enrollments of 19 and 28 students respectively. Three of eight programming assignments were adapted so that students submitted test sets along with their implementation.

It would be worthwhile to see a formal and controlled study of such an experience on a larger scale. Based only on our reflective statistics, we make some informal observations. Of 136 submitted programs, the combination of students' tests uncovered flaws in 80 of them (59%). The instructor performed independent testing, yet no additional programs were shown to be flawed; in fact, there was one program that succeeded in all of the instructor's tests yet was exposed by a student test. On average, a submitted test set exposed 74% of flawed programs. Not surprisingly, those students with correct implementation submitted better test sets as well. Test sets submitted by students with correct implementations exposed 87% of flawed programs on average, versus 63% for those test sets submitted by students with flawed implementations.

6 Automated Scoring

We have developed a Perl script, `autograde`, designed for handling the automated scoring of such an experience on a `Unix` system. Submissions are compiled and then executions are performed on test sets with the output captured to a file, which is then compared to the output of a correct implementation. Though not based on it, our high-level design is similar to the TRY system [17], albeit additional functionality automates the all-pairs competition. Other automated grading systems could likely be adapted for such a purpose.

We wish to share some insights, which may be particular to this purpose.

- *The scoring must be fully automated both in checking correctness and in compiling statistics.* Some testing systems are designed for interactive support when used by a grader [8]; such systems are not appropriate here. A fully automated system must be able to perform test after test without human intervention. A system must deal gracefully with programs which exit prematurely due to unexpected exceptions or which never exit due to infinite loops.

- *Scoring this experience is an offline, batch process.* Since students are submitting both implementations and test sets, the scoring process cannot be completed until all submissions are received. This is to be differentiated from testing systems that aim to give immediate feedback to students during development [2, 17]. Given the general turnaround time for grading assignments, it is quite reasonable if the scoring process needs hours or days to complete the thousands of tests involved.

- *The batch process should be recoverable and amendable.* Invariably, unforeseen errors may interrupt the scoring process. In such cases, the system should maintain enough log information to allow efficient continuation. Furthermore, if a late submission is accepted this results in the need both to test that student's implementation and to re-test other students' implementations on the additional test set. Similar re-grading may be needed if a student's source code is modified, either to remove spurious output or to benevolently fix a disastrous typo[2].

7 Availability

The `autograde` package of Section 6 together with complete information regarding the two examples of Section 4 will be made available at: `www.cs.luc.edu/~mhg/autograde/`

8 Acknowledgment

We thank Andy Harrington for incorporating such an experience into his own section of Loyola's data structure course and for subsequent conversations.

References

[1] ACM/IEEE-CS Joing Task Force. Computing Curricula 2001, Aug. 1, 2001. Steelman Draft.

[2] Baker, R. S., Boilen, M., Goodrich, M. T., Tamassia, R., and Stibel, B. A. Testers and visualizers for teaching data structures. In *Proc. 30th SIGCSE Technical Symp. on Computer Science Education* (New Orleans, LA, Mar. 1999), pp. 261–265.

[2] Most common, is the errant comment added hastily before the deadline, yet causing a compilation error.

[3] Gersting, J. L. A software engineering 'frosting' on a traditional cs-1 course. In *Proc. 25th SIGCSE Technical Symp. on Computer Science Education* (Phoenix, AZ, Mar. 1994), pp. 233–237.

[4] Goodrich, M. T., and Tamassia, R. *Data Structures and Algorithms in Java*, second ed. John Wiley & Sons, New York, 2001.

[5] Hilburn, T. Software engineering – from the beginning. In *Proc. Ninth SEI Conf. on Software Engineering Education* (Daytona Beach, FL, Apr. 1996), pp. 29–39.

[6] Hilburn, T. B., and Towhidnejad, M. Software quality: A curriculum postscript? In *Proc. 31nd SIGCSE Technical Symp. on Computer Science Education* (Austin, TX, May 2000), pp. 167–171.

[7] Hitchner, L. E. An automatic testing and grading method for a C++ list class. *SIGCSE Bulletin 31*, 2 (1999), 48–50.

[8] Jackson, D., and Usher, M. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Technical Symp. on Computer Science Education* (San Jose, CA, Feb. 27–Mar. 1, 1997), pp. 335–339.

[9] Jackson, U., Manaris, B., and McCauley, R. Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. In *Proc. 28th SIGCSE Technical Symp. on Computer Science Education* (San Jose, CA, Feb. 27–Mar. 1, 1997), pp. 360–364.

[10] Jones, E. L. Grading student programs – a software testing approach. *J. Computing in Small Colleges 16*, 2 (2001), 185–192.

[11] Jones, E. L. Integrating testing into the curriculum – arsenic in small doses. In *Proc. 32nd SIGCSE Technical Symp. on Computer Science Education* (Charlotte, NC, Feb. 2001), pp. 337–341.

[12] Kay, D. G., Isaacson, P., Scott, T., and Reek, K. A. Automated grading assistance for student programs. In *Proc. 25th SIGCSE Technical Symp. on Computer Science Education* (Phoenix, AZ, Mar. 1994), pp. 381–382.

[13] McCauley, R., Archer, C., Dale, N., Mili, R., Robergé, J., and Taylor, H. The effective integration of the software engineering principles throughout the undergraduate computer science curriculum. In *Proc. 26th SIGCSE Technical Symp. on Computer Science Education* (Nashville, TN, Mar. 1995), pp. 364–365.

[14] McCauley, R., Dale, N., Hilburn, T., Mengel, S., and Murrill, B. W. The assimilation of software engineering into the undergraduate computer science curriculum. In *Proc. 31nd SIGCSE Technical Symp. on Computer Science Education* (Austin, TX, May 2000), pp. 423–424.

[15] McCauley, R., and Jackson, U. Teaching software engineering early – experiences and results. In *Proceedings of the 1998 Frontiers in Education Conference* (Tempe, Arizona, Nov. 1998), pp. 800–804.

[16] Pargas, R., Underwood, J., and Lundy, J. Tournament play in CS1. In *Proc. 28th SIGCSE Technical Symp. on Computer Science Education* (San Jose, CA, Feb. 27–Mar. 1, 1997), pp. 214–218.

[17] Reek, K. The TRY system – or – how to avoid testing student programs. In *Proc. 20th SIGCSE Technical Symp. on Computer Science Education* (Feb. 1989), pp. 112–116.

[18] Robergè, J., and Suriano, C. Using laboratories to teach software engineering principles in the introductory computer science sequence. In *Proc. 25th SIGCSE Technical Symp. on Computer Science Education* (Phoenix, AZ, Mar. 1994), pp. 106–110.