Growing Testers: Incorporating Testing Concepts Throughout the CS Curriculum

Ronald Finkbine[1]
Department of Computer Science, Indiana University Southeast
New Albany, IN, 47250, USA

Peter Macpherson[2]
Department of Applied Technology, Rogers State University
Claremore, OK, 74017, USA

Abstract

Traditionally, software testing is introduced to students in Introduction to Programming, and then not treated in depth until an upper level course in Software Engineering. Software testing is often taught as a standalone subject instead of intertwined with all areas of software development. This treatment indicates to students that testing occupies a minor role in the field. This paper proposes an alternative approach of integrating testing methods progressively through the CS curriculum. As students master new CS materials, they will be exposed to the appropriate methods for testing their programs. In addition, this paper makes the claim that appropriate testing should be a distinct component in the grading of assignments.

1. SHOULD TESTING BE TAUGHT DISTINCTLY?

In the Denning Report (Denning 1983), the design of algorithms emphasizes the testing, as well as the implementation of algorithms. In the most recent ACM/IEEE Computing Curricula (Tucker 1991), testing is treated within knowledge units SE3 "Software Requirements and Specifications" and SE5 "Verification and Validation". Both of these topics are introduced earlier in the suggested CS1/CS2 courses as well as a specific Software Engineering course. "Validation" tests if the system performed according to the specification while "Verification" tests if the system performs according to the design. Clearly a consensus has formed on the importance of software testing as a subject of interest for academia and industry. Partnership between academia and universities have been established to encourage training in software engineering and testing as have been documented at the Software Engineering Institute (http). While academia needs to improve the teaching of software testing for industrial needs (Bach 1997), concentration on developing these skills in students improves the quality of instruction overall.

Too often, students immediately start coding once a problem given without fully understanding the problem. This behavior is sometimes inadvertently reinforced by the open-lab training methods expressed by many in the CS field. In such cases the requirements phase consisting of the written problem description, typically with sample input/output is given directly to the student. The design phase is rushed through as the student begins the implementation. If the final program executes correctly using the given sample input, the student considers the project a success. We believe the creation of the tests should be a distinct portion of the program grade to force the student to more fully examine the problem. Typically, the student must develop and submit their function tests prior to beginning to actual code the program. Once the program is completed, the student must show the results of executing their tests. This approach gives the student a more complete view of the software cycle.

2. PROPOSED TESTING INTEGRATION

Formalism has been introduced to the field of testing with the IEEE standard for unit testing (1008-1987) and testing documentation (829-1983). The field has matured to the point we can distinguish the types and purposes of tests. Figure 1 lists the category of testing and the appropriate class in which to integrate it within the CS curriculum.

3. HOW TO EVALUATE THE TESTS

Whether the labs are open or closed in CS1, a level of formalism should be introduced towards testing. While others have implemented degrees of formalism in CS1 courses (Levy), students have objected to the additional burdens placed upon them (Hilburn 1997). We propose

[1] rfinkbin@ius.edu
[2] macphersonp@acm.org

Test	Category	Class Introduced
System	Treats the entire system as a black box, not allowed to investigate within system	CS 1
Module/Unit	Performed on individual modules in isolation	Data Structures/CS 2
Integration	Checks the components work together collectively	Data Structures/CS 2
Environment	Test after porting to an environment other than on which the code was developed	Operating Systems
Glass Box	Test within programs, checks specific execution paths	Software Engineering
Acceptance	Satisfies the tests of the end user/customer	Senior Project

Figure 1. Testing Schedule

a simple implementation: the students should be required to submit their own test suites prior to coding of the program. This introduces the students to the idea of requirements as well as forcing a deeper understanding of the algorithms in use prior to the coding.

In order to coordinate test case summary and test case proof sheets, each test case should be numbered. Completed test cases require a proof sheet, which includes a printout of the test case input file, a copy of the output produced, and any hand-written explanations by the programmer/tester. This method requires the programmer to maintain, or save, multiple input files for successive testing during the life of a program to ensure consistency in the testing of multiple versions of the program source code. These efforts increase the likelihood of producing supportable software.

Performance of regression testing at the completion of development insures that any modifications required to pass later test cases did not introduce an error that causes the failure of a previously passed test case. An advanced test suite (or harness) would allow sequential running of all test cases, complete with separate input and output files for each test case. This test harness (usually a script or batch file) would allow for complete regression testing with only one input command from the programmer/tester.

4. DETAILED EXAMPLE

As a sample project, a data structures course was tasked to develop a bank queue simulation that will accept up to N tellers and has a line inside the branch for up to M customers. The customers arrive according to a Poisson distribution each with a randomized transaction time. The user provides the actual values of N and M. The program should find statistics about the system upon completion of the simulation, including teller idle time (efficiency), the average customer wait time, the minimum wait time, the maximum wait time, and the overall system throughput. In addition, the simulation should allow for customer balking upon entering the bank and also allow a customer to balk once they are in the wait queue.

First, the student should validate the simulation control values. Among the things to be checked are such diverse elements as: negative number of tellers, negative maximum number of customers, valid arrival rate for customers, and valid service times.

Next, the program should be tested with the common cases shown in Figure 2. Unlike the previous tests these are valid but often-problematic cases with the exception of case 22 which might be an error state. To test the rest of the system the Poisson arrival function should be circumvented to force the cases.

Most students would use *ad hoc* testing to validate their programs. Of those who bothered to test in a systematic manner, only a very small number would include all of Figure 2 test cases in a test suite.

5. BENEFITS OF TESTING

As discussed above, the traditional lecture method implies to the student that a successful programmer continually would ask the sole question, "Does this program calculate the answer I need?" The better question that a well-qualified professional would ask would be "Does this program calculate the answer I need, while continuing to detect common errors from all inputs? In addition, does it give full information to users so as to limit the number of times any user will call me, the programmer, late at night?" Programmers generally view software testing as a hurdle to be avoided instead of a path to better software.

Testing does not occur only on software. It can and should be used on all system deliverables at the conclusion of each stage of software development. Testing the requirements document helps determine if the student truly understands the problem. Testing at this stage illustrates the ambiguity of human languages in specifying a problem. Test definition at the design stage will force the student to define success—what exactly are the characteristics of successful software development? Post-coding testing enforces coding rigor; the student should have developed software that will execute correctly under all test cases posed prior to coding. In addition, it forces the student to constantly question the correctness of his code, thus enforcing that code is written with specific intent in mind, not

Number	Test Condition
Invalid 1	<1 teller
Invalid 2	Arrival rate <0
Invalid 3	Service time <0
1	1 teller, 0 customer (divide by 0)
2	1 teller, 1 customer
3	1 teller, 2 customers in sequence (serial) with lag between
4	1 teller, 2 customers serial with no lag
5	1 teller, 2 customers one exits exactly as one enters
6	1 teller, M customers mixed
6	2 tellers, 0 customer (divide by 0)
7	2 tellers, 1 customer
8	2 tellers, 2 customers serial with lag
9	2 tellers, 2 customers serial with no lag
10	2 tellers, 2 customers one exits exactly as one enters
11	2 teller, M customers mixed
12	N tellers, 0 customer (divide by 0)
13	N teller, 1 customer
14	N teller, 2 customer
15	N teller, N<M customer
16	N teller, N>M customers, no balking
17	N teller, N>M customers, balking at door
18	N teller, N>M customers, balking from within line
19	Shrink tellers by 1 (lunchtime)
20	Shrink tellers by 2 (lunchtime)
21	Shrink tellers by N-1 (lunchtime)
22	Shrink tellers by N (tellers on strike)
23	Grow tellers by 1 (after lunch)
24	Grow tellers by 2 (after lunch)
25	Grow tellers by N-1 (after lunch)
26	Grow tellers by N (after lunch)

Figure 2. Test Cases

randomly as many programmers (both student and professional) seem to do.

6. CONCLUSION

Standard open labs having students code too quickly encourages limited, short-term thinking. Tradition teaching methods for algorithm-familiarization, where a faculty member describes a problem, proposes and then codes a solution all within 50 minutes, reinforces the student's natural inclination to begin coding immediately. Many computer science curricula are moving toward a problem-solving approach to the introductory course. We recommend the first programming course, and the entire curriculum, be complemented with increased concentration on test case design.

7. AUTHOR INFORMATION

Dr. Finkbine is an Assistant Professor of Computer Science at Indiana University Southeast and has a Ph.D. in Computer Science from the New Mexico Institute of Mining and Technology. Dr. Macpherson is a Professor of Computer Science at Rogers State University and has a Ph.D. in Computer Science from Lehigh University.

8. REFERENCES

Denning, P. J., ed., 1983, "Computing as a Discipline", Communications of the ACM, 29, 3 (March 1983), pp. 9-23.

Tucker, A. B., 1991, "Computing Curricula 1991", Communications of the ACM, 34, 6 (Jun. 1991), pp 68 – 84.

http://www.sei.cmu.edu/collaborating/ed/ed.html

Bach, J., 1997, "SE Education: We are on our own", IEEE Software (Nov. 1997)pp. 26-28 .

Levy Kortright, L. M., "From Specific Problem Instances to Algorithms in the Introductory Course", SIGCSE pp.71-75.

Hilburn, T. M. and Towhidnejad, M., 1997, "Doing Quality Work: The Role of Software Process Definition in the Computer Science Curriculum", SIGCSE Bulletin, 29, 1 (March 1997), pp 277-281.