Teaching Software Testing in Introductory CS Programming Courses

Syed M. Rahman Akram Salah Mohammed Gomaa

Benzir M. Ahmed Anupam Kumar Nath

Department of Computer Science, North Dakota State University
258 IACC Building, Fargo, North Dakota 58105, USA
{Syed.Rahman, Akram.Salah, Mohammed.Gomaa,
Md.Ahmed, Anupam.Nath}@ndsu.edu

Abstract

Undergraduates in Computer Science (CS) typically begin their curriculum with a programming course or sequence. Many researchers found that most of the students who complete these courses, and even many who complete a degree, are not proficient programmers and produce code of low quality. In this paper, we try to address this problem by proposing a *cultural shift* in introductory programming courses. The primary feature of our approach is that software testing is presented as an integral part of programming practice.

Our initial results are that this approach improves students' program quality, in terms of black box testing. We found that teaching basic concepts of creating test cases and test the program do not take much time, it helps beginning students to understand the requirements, and it helps them produce better-quality code. Moreover, in our experiment, 67% of the students claimed that applying our approach makes the code easier to debug and improves the reliability and quality of their program.

1 Introduction

An industry survey [9] has reported that more than 50% of a software project's budget is spent on activities related to improving software quality. Industry leaders claim that this is caused by the inadequate attention paid to software quality in the development phase. Another multi-national, multi-institutional [1] assessment showed that students who completed one or two computer-programming classes' on average scored only 22.89 out of 110 points on the general evaluation criteria. Universities in USA, Canada, and elsewhere found that 50% of the students failed, withdrew or earned D-grades in introductory programming courses [12, 18]. These disappointing and alarming research results concluded that many students do not know how to program at the end of their introductory programming courses. This model of teaching makes the students unprepared to develop reliable software.

In this paper, we have addressed this problem and proposed two different models for two introductory programming courses. Our initial finding shows that our approach can be successful. We are running the experiment in the Department of Computer Science. In our first model, CS-I (Introduction to Java Programming), students write test cases as a prerequisite of writing programs. Students learn how to write test cases and how to test their own code. Students draw context diagrams, answer a few general questions, write test suites before writing code, and submit all these to the instructor. After writing code, students execute their test cases and submit test results and the test program with the main program. In our second model, CS-II (Data Structure using Java), students apply Test-driven Development (TDD) or test-first programming as a software development and testing methodology. In TDD, one always writes test cases before adding new code. It promotes incremental development and gives students a great degree of confidence in the correctness of their code, helps them understand the requirements and design better, makes it easier to change requirements, and helps to build reusable code [16, 17].

The rest of the paper is organized as follows. Section 2 shows some background studies and introduces TDD, section 3 explains the experiments, and section 4 enlightens the evaluation procedures. Section 5 presents our initial findings, and section 6 concludes with a discussion of our findings and a roadmap to future work.

2 Background Study

We believe that learning a computer program is a challenging task and requires a lot of hard work and dedication. Students often view the product of their labors as the executable program rather than the source code. Their focus is on writing a program that produces "correct output" for the required sample set of the assignment [14].

Gary Litvin [3] showed that the life expectancy of a working program is just a few months, whereas the source code, updated periodically, can live for years. Most likely, the programmer rewriting or reusing the code is not the original programmer. As a result, to be most useful the source code must be easily changeable. Easily maintainable code needs to be readable and coherently modularized into independent functions or classes that exhibit weak coupling and strong cohesion. Ideally, these programs also support encapsulation and information hiding. Comprehensive programming style and coding guidelines encourage the production of dependable and more easily maintained. This is not always the case but rather a frequent theme. [3].

2.1 Test-Driven Development

TDD has been popularized by extreme programming [11]. In TDD, one always writes a test case (or more) before adding new code. In fact, new code is only written in response to existing test cases that fail. TDD is attractive for educational use. It is easier for students to understand and relate to than many traditional testing approaches. It promotes incremental development and the concept of always having a "running (if incomplete) version" of the program at hand, and endorses early detection of errors introduced by coding changes. It directly combats the "big bang" integration problems that many students see when they begin to write larger programs, where testing is saved until all the code writing is complete. TDD boosts students' confidence in the portion of the code they have finished and allows them to make changes and additions with greater confidence because of continuous regression testing. Most importantly, students begin to see these benefits for themselves after using TDD on just a few assignments [16,17].

Unfortunately, in most undergraduate programs, students get little practical training in how to test their own code and often have poor skills (and even poorer expectations) in this area. In order to make a cultural shift in the way our students gain and apply testing skills, Edwards [17] applied TDD for developing students' programming assignments. Students who practice TDD produced 45% fewer defects per thousand lines of code, compared to another team who did not use TDD.

3 Experiment Setup

In our experiment, the same instructor teaches two sections of CS-I classes. The instructor teaches one section following our model where students write test suites as a prerequisite of the program and another section where students do not. Keeping all teaching materials, syllabus, grading policies the same, we find our model effective. We measure different metrics such as number of bugs found, complexity of the program, number of lines of code, number of methods used in the program, and the

time it takes for writing code and testing. In the next semester, we will run simultaneously our model-1 and model-2 in the same courses again.

In model-2, in programming class CS-II, one-section of students practice our model i.e. practicing TDD as a software development and testing methodology. Comparing one section's students who practice TDD with another section's, we measure TDD's effects on students' accomplishments. We also measure the effect on software quality in terms of fault density and other matrices such as number of lines of code, number of methods/modules used, and complexity of the program. We believe that the use of TDD is harder for students if they do not have at least some background in programming and testing. We chose course CS-II for applying TDD because students in this course already have sufficient background in programming including understanding programming requirements, design, and testing.

We understand that introductory computer programming classes already have a huge load. We would like to reduce the instructors' and graders' (or teaching assistants') pressure by developing an online evaluation tool. Using our tool, students upload their program online and validate their program. In addition this tool automates the grading strategy to be used to evaluate student written code and to provide clear, immediate feedback to students about the effectiveness and validity of their test suites. In the traditional grading system, generally students do not have any direct contact with their grader (typically a teaching assistants or a graduate student). In many cases, students do not even pick up their assignments and projects as long as they get scores. However, they end up making the same mistakes repeatedly. Students mainly focus on producing "correct" output and creating a version of compiled code. Moreover, a compiled code is not necessary enough and correct. Our under-developed interactive helping tool would reveal students programming errors, and identify problems or different solutions that would lead them to write better quality programs.

3.1 Writing and testing test cases for a sample problem

Here is a sample problem where we have applied our model. Students submit step 1 to step 4 before writing the program. They write the program, execute the test cases, and find the test results. Typically, students get a one-week window for submitting the compiled program, test program, and test result.

Step 1: Problem of statements: Write a program that asks user input for a Zip code and displays the corresponding bar code, or asks for bar code input and displays a Zip code. Print an error message if either the bar code or Zip code is not correct. The Zip code will be either 5 digits or 9 digits and bar codes will be either 32 or 52 bars [13].

(Use ":" for half bars, "|" for full bars. For example, the Zip code 95014 (Figure 2) becomes ||:|:::|:|:||::::::||:|::|:::|||)[1]

Step 2: Construct a context diagram (input-output chart):
In the Figure 1 shows the context diagram.

Figure 1: Context diagram for the sample problem.

Step 3: General questions

- What is the input data type (e.g. Integer, decimal)?
- What is the maximum and minimum value for the bar code and Zip code
- Can the input be zero or any negative values?
- Can the input be non-numeric data?
- Is there any possibility for Integer division or division by zero in the calculation?

Step 4: Generate Test Cases
Table 1 shows a sample test suite that students submit before writing code.

Table 1: Sample test cases that a student writes before writing code

Test Case	Input: Bar code or Zip code	Expected output	Valid/ Invalid input	Actual program output	Pass/ Fail
1	00000	|||:::||:::||:::||:::||:::||:::|	Valid		
2	|||:::||:::||:::||:::||:::||:::|	00000	Valid		
3	99999	||:|::|:|::|:|::|:|::|:|::|:|:|	Valid		
4	000000000	|||:::||:::||:::||:::||:::||:::||:: :||:::||:::||:::|	Valid		
5	999999999	||:|::|:|::|:|::|:|::|:|::|:|::|:|: :|:|::|:|::|:|::|	Valid		
6	123456	Invalid message	Invalid		
7	1234567890	Invalid message	Invalid		
8	1234	Invalid message	Invalid		
9	123D2	Invalid message	Invalid		
10	|||:::||:::::||::||::||::::::|||	03600	Valid		

Step 5: Execute the test cases (after writing the program and filling in the actual program output column)
Table 2 shows the executed results of the sample test cases.

[1] Please see the Appendix 1 for five digit encoding system. A correction digit follows five encoded digits.

Step 6: Result of the test cases (Filling in the pass/fail column)
Write the test cases result whether the test case passes or fails. Students submit this test execution result with the test program as well as the main program.

Table 2: Results of the test cases that students get after executing the program.

Test Case	Input: Bar code or Zip code	Expected output	Valid/ Invalid input	Actual program output	Pass/ Fail
1	00000	|||:::||:::||:::||:::||:::||:::|	Valid	|||:::||:::||:::||:::||:::||::|	Pass
2	|||:::||:::||:::||:::||:::||:::|	00000	Valid	00000	Pass
3	99999	||:|::|:|::|:|::|:|::|:|:::|:|:|	Valid	||:|::|:|::|:|::|:|::|:|:::|:|:|	Pass
4	000000000	|||:::||:::||:::||:::||:::||:::||:::||:::||:::||:::|	Valid	Invalid Message	Fail
5	999999999	||:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|	Valid	||:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|:|::|	Pass
6	123456	Invalid message	Invalid	Invalid message	Pass
7	1234567890	Invalid message	Invalid	(Valid bars)	Fail
8	1234	Invalid message	Invalid	Invalid message	Pass
9	123D2	Invalid message	Invalid	Invalid message	Pass
10	|||:::||:::::||::||::||::::::|||	03600	Valid	(Invalid bars)	Fail

3.2 Point distribution for grading

We recommend that the CS-I course instructor give the following sample point distributions. Students in one section are assigned testing as part of their requirements and students in another section are not. Table 3 shows a sample grading procedure for CS-I course:

Table 3: Sample point distribution for model-I (for CS-I course)

Items	Quantity	Each item without testing	Testing	Total
Exams	3	100	0	3X100+0 =300
Projects	3	40	3X10 =30	3X40+30 =150
Labs	6	20	6 X 5 = 30	6X20+30 =150
Total Points			**60**	**600**

Total testing points would be 60, which is 60/600 = 10 % of the total grade.

In Course CS-II, we give more emphasis on testing. The main concept in TDD is "write a little test, write a little code."

4 Evaluation Procedure

4.1 Teaching Software Testing

In the CS-I classroom, we explained how to write test cases. We spent only 25 minutes and showed one example of how to write test cases. Right after the presentation, we provided a similar problem and asked students to write test cases. More than 70% of the students came up with test cases. Of course, they do not know in detail about software testing or its different techniques. Our goal was to teach general concept and terminology of software testing so that a student would be able to create a simple set of test cases for a program comparable to those used in introductory programming projects.

4.2 Measuring Program Quality

In our experiment, the same instructor taught both section-1 and section-2 of CS-I class. We collected students' projects for both sections. Section-1 students did not follow our approaches, so they did not write test cases before writing code. Section-2 students did follow our approaches. They submitted the test suites one week earlier than their final submission of their project. We created a test suite following different testing techniques such as boundary value analysis and equivalence partitioning. We executed all test cases in all students' programs in section-1 and section-2. Executing and comparing the black box testing passing rate in both section, we found that model-1 was effective.

4.2 Conducting Surveys and Interviews

In our approach, students' involvement plays a vital role. We conducted a pre-test and a post-test survey of students' opinion and understanding about our model. End semester, students' feedback about our approach was very positive. 67% of the students strongly agreed or agreed that applying our approach made it easier to find bugs in their programs. In another question, 67% of the students claimed that writing the test suite before writing code improved the reliability and quality of their code. The rest of the students mainly expressed neutral opinions.

5 Results

In this paper, we propose two models in two introductory programming classes to improve software quality. Our approach is to make a *cultural shift* in teaching programming languages by making testing an integral part of programming practices. Students not only need to produce correct output but also need to understand how to test their code. In our opinion, if they know how to write the

program then they better know how to test it and make sure that their programs do what they expected to do.

We found that teaching basic concepts and terminology of software testing does not take much time. We spent only 25 minutes teaching students how to write test cases, and more than 70% of the students came up with test cases in the classroom. We collected students' projects and measured the quality of the program by applying the same test suites to both sections' code. Student feedback was very positive about our model. One student's instant comment in the classroom was, "It is something useful and certainly worthy to give a try."

We have not completed the experiment in this semester yet. We run the same experiment again next semester. Our initial finding shows that the passing rate of test cases is significantly higher for students who followed our approaches than for students in the other section. However, we need to run more projects/assignments in different programming courses before making any definite conclusions. We found that writing test cases before writing code helps students understand the problem better. Students like testing their code and it boosts their confidence.

Moreover, students' feedback about our approach was very positive. 67% of the students strongly agreed or agreed that use our approach makes it easier to find faults or failure in their programs. In another survey-question, 67% of the students claimed that writing test suite before writing code improves the reliability and quality of their programs. In both cases, the rest of the students mainly expressed neutral opinion about our approach. Although our initial assessment shows that our model is successful, we did not reach any definite conclusion yet.

6 Conclusions

Programming skills and writing good quality code or software are a common expectation for computer science students or for graduate computer science majors. Many researchers found that most of the undergraduate students who complete introductory programming courses, and even many who complete a degree, are not proficient programmers and produce code of low quality [9]. In this paper, we addressed this issue and proposed a *cultural shift* in teaching programming languages.

We proposed two different models for two introductory programming classes. In the first model, course CS-I, students would get a preliminary idea about how to write test cases and test program. In the second model, course CS-II, students would practice TDD as their programming methodology and testing approach. We believe that testing should be an integral part of students' programming practices. It is not enough to produce a compiled version of code and correct output; students need to test their program accordingly.

In our experiment, the same instructor teaches the same course in two different sections to eliminate many factors that vary from instructor to instructor. The instructor in one section followed our model i.e. apply testing as a prerequisite of writing code and another section did not. Keeping all factors the same, we measured how our model can play a role in improving programming quality.

Our initial experimental result shows that our model is successful. We found that our approach helps understand the problem and improves software quality (in terms of black box testing). We also found that teaching basic concepts of software testing to beginner students does not take much time, and student feedback was very positive. In the next semester, we will apply our approach in different introductory programming courses.

Acknowledgement

The authors acknowledge the contributions of the course instructor Dr. John Martin who has involved throughout the whole experiment.

References

[1] McCracken, M., Almstrum, V., Laxer, C., and others, *A Multi-national, multi-institutional study of assessment of programming skills of first year CS students*, volume 33, issue COLUMN: ITiCSE 2001 working group reports, Pages: 125 – 180, 2001

[2] Truong,N., Roe, P., Bancroft, P., *Static Analysis of Students' Java Programs*, ACM International Conference Proceeding Series, Proceedings of the sixth conference on Australian computing education - Volume 30, Dunedin, New Zealand, Pages: 317 – 325, 2004

[3] Lionel, D., Pozefsky, P., *Implementation of Programming Standards in a Computer Science Department*, Proceedings of the 17th Annual Southeast Regional Conference. April 1979. p. 142-143.

[4] Litvin, G., *The 32 Bits of Style*, www.skylit.com/style/index.html.

[5] Ostrand, T. J., Weyuker, E. J., *The Distribution of Faults in a Large Industrial Software System*, ACM SIGSOFT Software Engineering Notes, Volume 27 , Issue 4 July 2002

[6] Juristo,N., Moreno, A., Vegas, S., *Reviewing 25 Years of Testing Technique Experiments*, Empirical Software Engineering, 9, 7–44, 2004.

[7] Adams, E.N., *Optimizing Preventive Service of Software Products*, IBM J. Res. Develop., Vol28, No1, Jan 1984, pp.2-14.

[8] The Software Engineering Institute (SEI), http://www.sei.cmu.edu/sei-home.html, (November 10, 2004)

[9] Townhidnejad, Hilburn, *Software quality: a curriculum postscript?*, Technical

Symposium on Computer Science Education, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, Austin, Texas, United States Pages: 167 – 171, 2000

[10] Andy Hunt, A., Thomas, D., *Pragmatic Unit Testing in Java with Junit*, O'Reilly, ISBN: 0-9745140-1-2, September 2003

[11] Extreme programming, http://www.extremeprogramming.org/, (November 10, 2004)

[12] Nagappan, N., William, L., Ferzil, M., Wiebe, E., Yang, K., Miller, C., and Balik, S., *Improving the CS1 Experience with Pair Programming*, Proceedings of the 34th SIGCSE Technical Sympoisum on Computer Science Education, Reno, Nevada, USA, March 2003.

[13] Horstmann, C., *Computing Cencepts with Java Essentials*, 3rd Ed, John Wiley & Sons, Inc., NY, USA, Page: 326-328

[14] Shepard, T., Lamb, M., Kelly, D., *More testing should be taught*, Communications of the ACM, Volume 44 , Issue 6 (June 2001), Pages: 103 - 108

[15] Allen, E., Cartwright, R., Reis, C., *Production programming in the classroom*, Technical Symposium on Computer Science Education, Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, Navada, USA, (2003), Pages: 89 - 93

[16] Edwards, E., *Using test-driven development in the classroom*, Providing students with concrete feedback on performance. In Proceedings of the International Conference on Education and Information Systems: Technologies and Applications (EISTA'03). August 2003

[17] Edwards, E., *Teaching software testing: Automatic grading meets test-first coding*. In Addendum to the 2003 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications. 2003.

[18] Hermann, N., Popyack, J., Char, B., Zoski, P., Cera, C., and Lass, R.N., *Redesigning computer programming using multilevel online modules for mixed audience*, Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, Reno, Nevada, USA, March 2003.

Appendix 1

Postal bar codes. For faster sorting of letters, the USA postal service encourages companies that send large volume of mail to use a bar code denoting the Zip code. The encoding scheme for a five-digit Zip code is shown in the Figure 2. There are full-height frame bars on each side. The five encoded digits are followed by a correction digit, which is computed as follows: Add up all digits, and close the correction digit to make the sum a multiple of 10. For example, the Zip code 95014 has sum of digits 19, so the correction digit is 1to make the sum equal to 20. Each digit of the Zip code, and the correction digit, is encoded according to the Table 4 below.

Table 4: Encoded value for Zip code and correction digit

	7	**4**	**2**	**1**	**0**
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	1	0	0	0	1
8	1	0	0	1	0
9	1	0	1	0	0
0	1	1	0	0	0

Encoding for Five-digit Bar Codes:

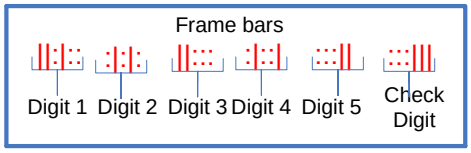

Figure 2: Five-digit US postal bar codes