INTRODUCTORY PROGRAMMING, PROBLEM SOLVING AND COMPUTER ASSISTED ASSESSMENT

Charlie Daly and John Waldron

Introductory Programming, Problem Solving and Computer Assisted Assessment

Charlie Daly
Computer Applications
Dublin City University
Dublin 9

John Waldron
Dept. of Computer Science
Trinity College Dublin
Dublin 2

cdaly@compapp.dcu.ie

Abstract

Constructing a mathematical proof is isomorphic with writing a computer program. Validating a proof is as difficult as validating a computer program. Yet there is a pragmatic approach to validating a computer program: simply testing it with sample input and checking the required output. This paper discusses the use of computer assisted assessment on an introductory programming course. It is formative assessment; the aim being to foster a problem solving approach to programming. We have discovered the following advantages with computer assisted assessment in our programming course: the students get plenty of practice in programming, they get fast feedback and can easily move onto more advanced problems as their skill develops.

The paper looks at the issues involved in problem design, the importance of presentation of the assessment and feedback, and student impressions of the exercise. Results are compared with traditional paper based examinations.

Introduction

There is a serious problem in introductory computing courses. The evidence is that the majority of students simply don't learn to program at the expected skill level (McCracken 2001). This raises a few questions, in particular, how do these students pass the programming exams.

To examine this question we need to understand what exactly is programming skill and how can it be developed. Most researchers in the programming languages' community understand that programming is a complex intellectual activity based on deep logical principles, but their insights are not being used to inform the introductory curriculum (Felleisen 2002). Mathematical problem solving is a significant part of the programming process. In fact, the seminal book on mathematical problem solving, "How to Solve it", by George Polya "sheds more light on the *activity* of software design (as opposed to the result) than any software design book I've seen" (McConnell 2002). Polya's techniques have been adapted to the programming process for use in introductory programming courses (Thompson 1997).

Students learning to program must master the syntax of a programming language and also develop program design and problem solving techniques. It seems as if many students can manage the programming language syntax but are not developing the requisite problem solving abilities. Despite this many of them are able to proceed beyond the introductory programming sequence. This obviously indicates that there is a serious problem with assessment. Generally, the assessment of programming skill consists of programming assignments and written exams. The assignments are subject to plagiarism and it is difficult to manually assess programming ability in standard written exams. This is partially a result of the simultaneous attempt to measure syntax and problem solving ability. Another factor is that the assessor can often see how a weak attempt could be easily improved to produce a solution. The student then gets credited with foresight that frequently isn't there. Ironically, human insight results in poorer assessment.

It is relatively easy to solve the problem of poor syntax. If the student has access to a computer (and in particular a compiler) then the student can use the compiler to locate syntax errors. A competent student can then correct those errors.

The second problem is more challenging. Unless the student's program works, it can be very difficult to determine if the student was progressing towards a solution. Certain kinds of programming problems can be tested by supplying input and observing the output that the program produces. If we provide a mechanism for students to test their programs, a student will be able to see if their program does not work and get a chance to correct the program. The resulting programs are much easier to assess and in fact it makes sense to assess them automatically.

Note that you can show the student what their program should have output without showing them the solution. This means that you can allow students to resubmit without penalty.

We have implemented a system, RoboProf, which manages the entire system in a user-friendly way.

RoboProf is in use in Dublin City University. The Computer Applications department has a one semester (12 week) introductory programming course (CA165). There are 370 students, of whom 300 are computing students, the remainder being maths students. There is a written exam (week 16) and two lab-based programming exams managed by RoboProf (at week's 7 and 12). This paper discusses the use of RoboProf on the course.

The Importance of Good Assessment

If we want the students to develop problem solving skills then we must properly assess these skills (and let the students know that we will assess it).

"The spirit and style of student assessment defines the de facto curriculum" (Rowntree 1977).

Once students know that they will need to be able to program in order to pass the exams, they will be interested in developing a programming skill. We have weekly lab work that is not directly assessed. Yet the majority of the students do this work and will finish it in their own time if they don't manage to complete it in the lab. Students realise that they must learn to program and therefore there is no point in avoiding the lab work. In other words, the formative assessment works because the summative assessment (the lab exams) work.

The students regard the two lab exams as summative assessment (each exam contributes 15% to the final grade), but we try and ensure that it is also formative. Some students begin to understand how to problem solve when forced to, i.e. when a student is alone with the problem and has only time and the computer to help solve it. We try and create an environment conducive to thinking (i.e. we minimise distractions). The lab exam is two hours long and students have three problems to complete. We give them a paper version of the problems and don't allow them to use the computer for the first half-hour. This should encourage the students to read the problem and design a solution. It also means that the first half-hour is quiet.

The exam is not open book. We suspect that having books available may not help weaker students. These students are likely to waste time looking for the solution in the book. Instead we allow all students to bring in a handwritten A4 'cheat-sheet' which can contain whatever they want. The process of creating the 'cheat-sheet' may also be educational.

The students have a practise exam the week before the real exam so that they can understand how the process works.

After the exam, the problems are discussed and we show the students how we would use standard problem solving techniques to approach the problems.

Although we wanted to create a friendly environment we had to be aware of security. To ensure that no student could cheat, we used standard security mechanisms. We created special accounts for the students so that they could not access their normal accounts. The computers are 1.5 meters apart and so it would have been obvious if they were trying to look into a neighbour's computer. Students were randomly allocated to computers so they were unlikely to be able to make arrangements with neighbours. Computer-savvy invigilators discreetly patrolled the lab to ensure that students only used the requisite software (a web browser, editor and compiler).

All the student programs were stored on the server and could be later checked (in case students tried to fool the system).

The programming problems need to be designed for automated testing. The problem specification needs to be very clear and it is a good idea to show a sample execution of the program. The first lab exam (at week 7) is shown in

Appendix A. There are three questions in order of difficulty. The first two are relatively easy and the last is a bit more challenging. Over 90% of the class solved at least one problem, 70% solved two and 25% solved all three. This means that 90% of the class can create and compile a simple program.

The second lab exam (week 12) had two questions (see appendix B). These questions were more difficult and the students did not do as well. Over 50% of the students got one problem completely correct and 15% got both problems correct.

RoboProf Implementation

RoboProf is a computer program which presents problems via the web, accepts student solutions, marks them and provides feedback to the student. When a student logs on; they are shown the problems. When a student has designed and tested a solution, they can submit the solution to RoboProf for testing. RoboProf simply runs the program against a prepared set of input data and compares the program's output with the expected output. The student is given a mark and shown the result of running their program for each test case alongside the expected output of the program. If the student doesn't get full marks, they can inspect their program output and the expected output. If the student spots an error in a program, then it may be corrected and resubmitted without penalty. This means that trivial output errors may be immediately corrected.

More details about RoboProf can be found in (Daly 1999).

Comparison with final written exam

The way the university's semester system is implemented encourages us to have a written exam. We compared the results of the programming exam (combination of both lab exams) and the written exam. Looking at Figure 1, we can see that the results in the programming exam match those in the written exam quite closely.

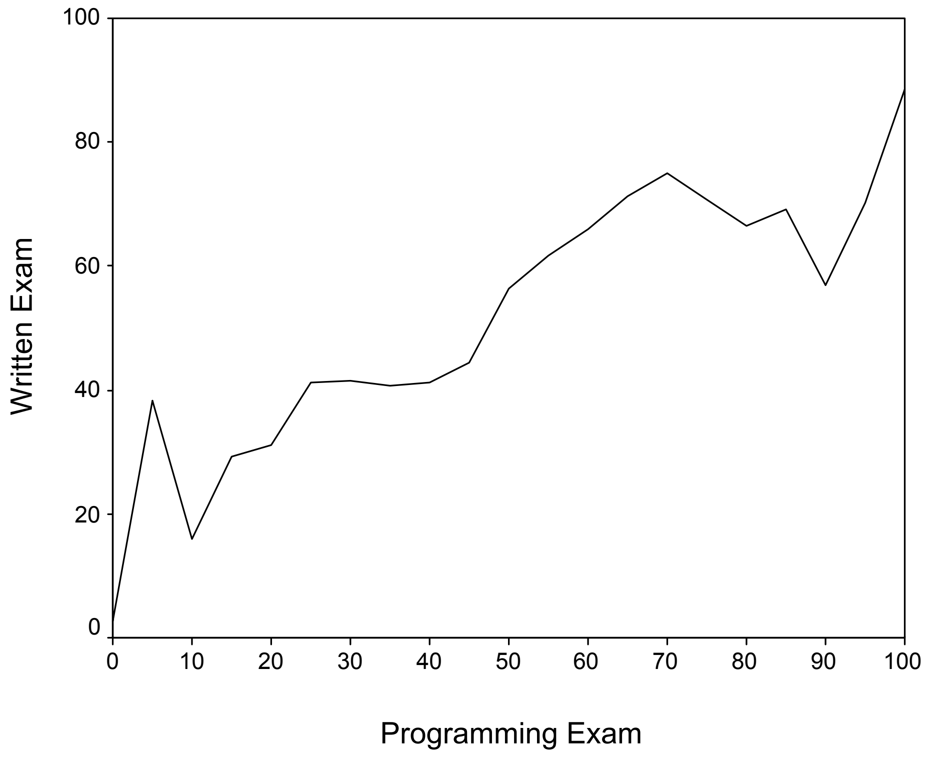

Figure 1. Graph of written exam Vs the programming exam

The marks of the programming exam are rounded to the nearest 5%. In each 5% band, the average of the marks in the written exam are shown. In particular, students who got less than 40% in the programming exam averaged less than 40% in the written exam. Students who got more than 40% in the programming exam averaged more than 40% in the written exam.

One might contend that this contradicts our earlier assertion that programming exams are better at assessing programming ability than written exams. However, the examiners were instructed to only award marks to programs that definitely looked as if they could work. In addition, because students had learned that only working programs received marks, they had become better at designing programs that worked. Working programs are much easier to mark.

There were some disadvantages with the programming exam. The programming problems and the software have to be well tested to ensure that there will be no technical hitches. And there are resource problems, using almost 400 of the department's computers at one time does not make us popular with other users. But of course, there are advantages. These exams don't need to be manually corrected. Four examiners required three days to correct the 350 written scripts.

Feedback from students

We gave the students an anonymous questionnaire after the programming exam (see appendix C). 245 students completed the questionnaire. The diamonds show the response which was closest to the average. The only surprising result was that although the students did not think that the marking system was fair, they did like the web based test. This was true even of the students who received no marks in the exam (though only just).

If you compare the response based on the number of problems the students received credit for, the results are unsurprising. Students who did better in the exam liked it more, thought it was fairer and thought their result reflected their ability.

The sound of other people typing was not perceived to be a distraction. However, one of the comments suggested that students leaving early were a distraction. If a student got all three problems correct, they tended to leave immediately. This could understandably shake the confidence of a student who was struggling with the first problem.

The marking system may be 'fixed' by selling it differently. Students don't like being told that they got zero marks for a program that nearly works. Instead the message should say that the program output doesn't match and politely suggest that they try again.

Conclusions

It is difficult to assess programming ability in written exams. Lab based exams where the student develops programs on a computer are fairer to the student and are easier to correct. In fact they can be successfully corrected automatically.

Although validating a computer program is difficult, simply checking expected output gives us confidence that the program works. By allowing students to resubmit programs, we bypass the problem where the program's output doesn't exactly match the required output.

Computer Programming is a creative process that belongs to the synthesis level of Bloom's taxonomy (Bloom 1956). As such it is a higher order skill that can be assessed using computer assisted assessment.

The only concern is that the students don't like the marking system. The feedback system needs more work to make it acceptable to the students. Students need to be made aware that a process of feedback is involved, where they are being helped to develop their programs. Currently they view the feedback as telling them that their programs are no good. We are working on the presentation of the feedback to counter this problem.

References

Bloom, B. S., et al (1956) *Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook I: Cognitive Domain* New York: Longmans, Green.

Daly, C. 1999 *RoboProf and an introductory programming course*, Proceedings of ITiCSE '99 ACM Press.

McCracken, M et al. (2001) *An international multi-institutional study of introductory programming courses* Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS students.

McConnell, S (2002) http://www.construx.com/stevemcc/rl-top10.htm

Felleisen, M. (2002) *From POPL to the Classroom and Back.* Proceedings POPL 126-127.

Polya, G. (1957) *How to Solve it*, *2nd ed*., New York: Doubleday.

Rowntree, D. (1977) *Assessing Students: How shall we know them* Kogan Page 1

Thompson, S. (1997) *Where do I begin? A problem solving approach to teaching functional programming* In Krzysztof Apt, Pieter Hartel, and Paul Klint, editors, First International Conference on Declarative Programming Languages in Education. Springer-Verlag.

Appendix A: First Lab Exam

Question 1

Write a program that asks for a price and makes an assessment: if the price is more than £100.00, then print "too high", if it is less than £50.00 print "that's cheap" otherwise print "OK"

A sample run might be

```
What is the price?
120.99
too high
```

or

```
What is the price?
51.50
OK
```

Question 2

You are requested to write a program for proud users; it should read a number *num*, and a name (a String) and prints the name *num* times.

A sample run might be

```
Enter a number and name: 4 James
James
James
James
James
```

I.e. there **James** is printed 4 times.

Question 3 (Haggling)

A street seller needs a program to help him haggle. He only sells one item, worth £100, as soon as he's offered this (or more) he accepts the offer. If he's offered less, he demands a price such that the average of the offer and the demanded price is £100. He keeps this up until a deal is done.

A sample run might be

```
What is your offer?
400
It's a deal!
```

or

```
What is your offer?
50
I'll sell it for 150. What is your offer?
90
I'll sell it for 110. What is your offer?
100
It's a deal!
```

You can assume that a deal is always done. Note that the first demanded price was 150. This was in response to the offered price of 50 (the average of 150 and 50 is 100).

Appendix B: Second Lab Exam

Question 1

Write a program that reads in a sequence of integers and calculates the average, then prints out how many elements are greater than the average.

A sample run might look like:

```
How many numbers
7
Enter 7 numbers:
-6 8 -17 67 10 -7 0
3 numbers are greater than the average.
```

Note that the average is about 7.9, the elements 8, 67 and 10 are greater than that.

Question 2

Write a program that finds the tags in a String of characters. Print each tag on a separate line. Tags are contained in braces '{' and '}'. You may assume that the String is well formed, i.e. all braces match.

A sample run might look like:

```
Enter a string: Here is a tagged String {one} and {another}.
The tags are
one
another
```

Hint: You may find the following String methods useful.

- `int indexOf(int ch)` Returns the index within this string of the first occurrence of the specified character.
- `int indexOf(int ch, int fromIndex)` Returns the index within this string of the first occurrence of the specified character, starting the search at fromIndex.
- `String subString(int beginIndex, int endIndex)` Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

The simplest way to read in a String is to use the Console.readString() method.

Appendix C: Programming Exam Questionnaire

	Strongly Agree				Strongly Disagree
I liked the web-based test	☐	◆	☐	☐	☐
Web-testing is more efficient than a normal paper-based test	☐	◆	☐	☐	☐
I would feel more relaxed using paper-based test	☐	☐	☐	◆	☐
The automated marking system was fair	☐	☐	◆	☐	☐
The sample exam was useful	☐	◆	☐	☐	☐
The exam result reflected my programming ability	☐	☐	◆	☐	☐
I was surprised at how well I did	☐	☐	◆	☐	☐
I was surprised at how badly I did	☐	☐	☐	◆	☐
It would have been better if I could have used the computer in the first half hour.	☐	☐	☐	◆	☐
The typing sound made by other people made me nervous.	☐	☐	☐	◆	☐

How many completely correct problems did you get credit for

○ 1 ○ 2 ○ 3

Comments?

Thank you for completing the questionnaire.