What Are We Doing When We Assess Programming?

Dale Parsons
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Dale.Parsons@op.ac.nz

Krissi Wood
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Krissi.Wood@op.ac.nz

Patricia Haden
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Patricia.Haden@op.ac.nz

Abstract

Considerable research has been devoted in recent decades to identifying optimal pedagogical strategies for teaching computer programming. Often the result of these efforts has been to conclude that we have made little progress, as our students consistently perform poorly on the assessments we apply to measure teaching efficacy. In this paper, we suggest that a significant contributor to this poor performance may be the methods of assessment, which do not reflect the knowledge and skills that a real programmer needs to write real code. We propose an alternative assessment format using Activity Diagrams that better reflects true programming ability. Our preliminary results indicate that these assessments correlate well with the ability to produce working code, while more traditional question formats do not.

Keywords: Programming Education, Programming Assessment.

1 Introduction

In her seminal 1999 paper, Sally Fincher asked "What are we doing when we teach programming?" (Fincher, 1999) Fincher noted that there had been a shift in the perception of computer programming from a mechanical skill to an essential element of computer science theory, without a consensus on the implications of this shift for pedagogy. This discussion helped to launch CS Education as a formal area of academic research.

In the intervening 15 years there has been extensive exploration of approaches to the teaching of programming, comparing languages, tools and conceptual methodologies (e.g. Sajaniemi, Kuittinen and Tikansalo, 2008; Pears, Seidman, Malmi, et al., 2007).

Unfortunately, the most consistent conclusion drawn from this body of research is that in spite of all this effort we still don't know how to teach programming, because large numbers of our students fail our introductory programming courses (Bergin and Reilly, 2005; Gonzalez, 2006; Lahtinen, Ala-Mutka and Järvinen, 2005) and even those who pass don't seem to be able to program very well (Ford and Venema, 2010; Thomas, Ratcliffe, Woodbury, et al., 2002; Bornat, Dehnadi and Simon, 2008).

It seems illogical that after so much study, and after having produced a sufficient number of skilled programmers to drive the computing industry beyond recognition in the last two decades, we still conclude that our students can't program. Perhaps an alternative is possible: we're teaching successfully -- but we're assessing badly. Specifically, when we assess, we are confusing programming with the abstractions of computer science and symbolic manipulation. Students fare poorly on our assessments not because they can't program but because we are not testing their programming ability. A cow makes a terrible racehorse, but that doesn't mean she's not a very good cow.

1.1 The Role of Assessment

Accurate assessment of programming ability has multiple roles in programming education and programming education research. Primarily, of course, we wish to measure programming ability in the classroom to evaluate and rank students, deciding who passes and who does not pass our programming courses. In programming education research, we also use measures of programming ability as our dependent variables. We may, for example, wish to evaluate a teaching intervention by comparing students' programming ability with and without the technique. This requires an accurate measure of individual programming ability.

Often these two roles overlap. For example, in studies of teaching efficacy, researchers often use final course mark as a reflection of programming ability. Cardell-Oliver has clearly articulated the potential weakness of this approach, (Cardell-Oliver, 2011) but it remains extremely common (cf. Clear, 2008).

Thus our ability to accurately measure an individual's programming skill underpins both our educational and scientific endeavours.

The critical nature of assessment in both of these roles -- teaching and research -- is recognised by the CS Education community. Simon, Sheard, and their colleagues (e.g. Sheard, 2012; Simon, Sheard, Carbone, et al., 2012) have undertaken exhaustive descriptive studies of the types of questions used in examinations in programming courses. Various authors have carefully mapped individual exam questions to the Bloom and SOLO taxonomies (e.g. Lister, Simon, Thompson, et al., 2006; Whalley, Lister, Thompson, et al., 2006) in order to determine more precisely what is being tested by programming exams.

Discussion of metrics in research is also vigorous. Considerable debate has touched on whether McKracken's seminal "our students can't program" study (McCracken, Almstrum, Diaz, et al. 2001) was biased by

the use of an invalid metric of programming ability (Utting, Tew, McCracken, et al., 2013; McCartney, Boustedt, Eckerdal, et al., 2013). An assortment of tools have been proposed, explored and validated for use in research studies of programming education.

Study of this body of research gives clear insight into ***how*** we are assessing. It is our contention, however, that further analysis is required into ***what*** we are assessing.

1.2 Programming or Computer Science?

Fincher (Fincher, 1999) originally made the distinction between programming as a means to an end, and programming as a component of a theoretical discipline. She noted that before Computer Science existed as the unique theoretical discipline underpinning computation and computation systems, practitioners of engineering, chemistry and mathematics used programming as a way "to get the computer to do something". As Computer Science matured, programming became an area for theoretical exploration in its own right in the contexts of automata theory, language design, and compiler construction. It remained, however, the essential tool for getting computers *to do things*. Since 1999, computers and computing have become so ubiquitous that in nearly every academic and commercial discipline, getting computers to do things is indispensable. Not only the software development industry, but many associated disciplines that rely on computer software require people with advanced abilities in, specifically, computer programming. We contend then, that Fincher's maturation process has come full circle -- we now have two fully mature, separate but intertwined disciplines: computer programming and theoretical computer science. Computer programming is the ability to produce working digital artefacts to the standards dictated by industrial best practice. Computer science is the study of the underlying principles of computing and computation.

It is possible to find tertiary degree programmes that quite clearly direct their students in one or the other of these disciplines. One group seeks to produce graduates who are prepared to step into the information technology industry as software developers versed in the skills and techniques required of a professional programmer. The other seeks to prepare students for further study in experimental and theoretical areas of computer science. Both of these disciplines are relevant and challenging and, while they share many fundamentals, they clearly place different emphasis on the elements of computer science.

We believe that in many attempts to assess programming ability, both as an educational evaluation and as a performance metric in quantitative research, these two disciplines are becoming entangled. For example, Lister and his colleagues (Lister, Adams, Fitzgerald, et al., 2004) describe a suite of multiple choice questions that can be used to evaluate a student's ability to read code, an essential part of the ability to program (i.e. to generate working digital artefacts with a programming language). Among the questions is the following code fragment:

```
int[ ] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;

while (i < j)
{
        temp = x[i];
        [i] = x[j];
        [j] = 2*temp;
        i++;
        j--;
}
```

Students are asked to identify the contents of the array x after these statements are executed. This code fragment is written in syntactically correct Java and it is semantically coherent, but it isn't a piece of code that a professional programmer would ever write. It performs no logically delineated task; it provides no context for the purpose of the computation. Not being able to answer this question doesn't necessarily demonstrate that one is unable to write code to iterate over and modify the contents of an array, it merely indicates that this abstract and tortuous piece of code is difficult to understand.

Similarly, Dehnadi (Dehnadi, 2006) presents a suite of questions to test understanding of the concept of assignment in programming. Among the Dehnadi questions is this one:

```
int a = 5;
int b = 3;
int c = 7;
a = c;
b = a;
c = b;
```

The student is asked the values of a, b, and c after execution of these statements. Again, this is syntactically correct code, and it is certainly possible to work out the values of the variables. But it is very difficult to come up with a realistic scenario under which a programmer would actually write this fragment.

Questions such as these therefore attempt to test the ability to perform the act of programming by requiring an understanding of something that would never be done while performing that act. Students traditionally score very badly on these questions (the question from Lister had only a 73% success rate, even though it was a 5-item multiple choice, and this was actually the highest success rate of the entire suite). Under the assumption that these questions are testing the ability to program, we conclude, dispiritedly, that our students cannot program. However, if we accept that the ability to program is the ability to write digital artefacts to an industrial standard, this pessimism may be unwarranted. Since no one would need to produce such code to be a capable programmer, our students' failure to correctly answer these questions does not mean that they cannot program. It simply means that they cannot do whatever it is that these questions require.

This disconnect can be seen in authors' descriptions of their measurement tools. For example, Ford and

Venema's oft-cited work on assessment (Ford and Venema, 2010) is entitled "Assessing the Success of an Introductory Programming Course". However, the authors state quite specifically that they are not trying to assess the ability to program as we have defined it (i.e. the ability to produce working digital artefacts following current standards of best practice). Instead, they propose a suite of tests "to examine whether students who have passed an introductory course have achieved an understanding of fundamental concepts in programming" (pg. 1). That is, they wish to measure the mastery of fundamental concepts, rather than the specific skill of programming. Note that the former is not sufficient to demonstrate the latter -- that is, I may have a detailed understanding of the physical principles of flying, but still not be able to safely launch myself into the air. One might argue that mastery of fundamental concepts is at least necessary to demonstrate the skill of programming, but this presupposes that what has been identified as fundamental concepts are, in fact, fundamental and, more critically in this context, that they are being tested in ways that are relevant to the skill we are attempting to quantify.

Ford and Venema focus specifically on the concepts of "assignment and sequencing". These are clearly fundamental to the act of programming because modern programming languages provide constructs for both assignment and sequencing and all digital artefacts of nontrivial complexity must include these elements. Ford and Venema explore a number of existing metrics which test comprehension of the principles of assignment and sequencing, but which do so at a very high level of abstraction. For example, in addition to the questions of Dehnadi described above, and a similar suite from Ma (Ma, Ferguson, Roper, et al., 2007), they include the "Reges question":

> *If b is a Boolean variable, then the statement*
> *b = (b = = false);*
> *has what effect?*

This item certainly requires an understanding of the rules of assignment and sequencing, but it is not typical of the constructs working programmers ordinarily produce and thus, we suggest, does not pragmatically test the ability to program.

Ford and Venema observed generally poor performance on all of their metrics and concluded "that many students who had passed an introductory programming course had little or no understanding of fundamental concepts" (pg. 1). We would suggest that this conclusion does not necessarily follow. What one can conclude from the poor performance on such tools is that many students who had passed an introductory programming course were unable to solve these complex and abstract problems involving the assignment operation. They might still understand the fundamental concepts of programming, and perhaps, be able to program quite adequately for their level of experience. Being able to solve complex, abstract symbolic manipulation exercises is an extremely valuable skill, and one a person might certainly wish, in some circumstances, to be able to measure, but it is not equivalent to the skill of programming.

1.3 How to Measure the Ability to Program

We have argued that often, the traditional pencil-and-paper approaches to programming assessment (particularly code reading problems) are presented at a level of abstraction that makes them poor indicators of the ability to actually write good, working code. We need an alternative method to assess programming ability. The obvious suggestion is to have students create digital artefacts and assess them for functional accuracy and code quality via industrially approved metrics. We do, in fact, contend that this should be the Gold Standard for judging one's ability to program. Unfortunately, in the practical contexts of both student evaluation and educational research, this approach has its own significant shortcomings.

First, in many situations, using artefacts for assessment is prohibitively expensive. In large classes, it may simply be impossible to hand mark the large number of pieces of code that would be required. Even in our own institution, where courses are limited to no more than 48 students per semester, marking of coding assignments is a huge burden on teaching staff. Promising work is being done in the area of automated assessment of code (Ihantola, Ahoniemi and Karavirta, 2010), but it is not a problem that has yet been completely resolved. The same objection would apply to any research study that wished to incorporate artefact assessment as a measure of efficacy.

Second, in the classroom situation, issues of authorship arise when assessment is performed outside of controlled examination conditions. Many educators have noted concerns about plagiarism in student coding projects. While there are some tools that can be used to help detect plagiarism (e.g. Vamplew and Dermoudy, 2005) it again makes artefact marking a questionable choice, at least as the sole means of evaluation.

Third, artefact evaluation is impractical until actual artefacts can be produced, that is, until a certain level of coding skill has been attained. In the face of increasingly compelling evidence for the importance of effective pedagogy and the detection of difficulties in the very earliest weeks of programming education (Robins, 2010), it would be very risky to delay evaluation until students are experienced enough to write substantial pieces of software.

There are two steps that can be taken to ameliorate these difficulties. Most obviously, in a classroom situation, one can compose a course mark from a mixture of code artefacts and assessments performed under exam conditions. In our own introductory programming courses, we use a combination of code projects developed outside of class, written examination, and practical coding exercises conducted under examination conditions.

Further, when conducting written examinations, one can attempt to construct questions which reflect, as accurately as possible, skills used by *real programmers* in writing *real code*. To this end, it can be helpful to refine our notion of the fundamental concepts and capabilities of computer programming.

1.4 Programming Fundamentals

We have argued for computer programming as a profession distinguished from theoretical computer science, based on the ubiquity in the modern world of computer artefacts and computation. A related argument is increasingly made for not only the act of programming, but the act of "thinking like a programmer." The term generally used in this discussion is "computational thinking" (Wing, 2006). Although there is some variability amongst authors, the basic tenet of the computational thinking movement is that computers are used as aids to problem-solving in most professions, activities and endeavours in modern Western society. Further, that there is a common underlying style for solving problems with a computer based on problem decomposition and algorithmic construction -- effectively translating our human solutions into something a computer can do. It has been argued that being able to perform this style of problem solving, i.e. being able to think computationally, is as integral in modern society as literacy and numeracy. Computer programming -- by definition, getting a computer to do things -- naturally exhibits the computational thinking template. In this context, we view the process of producing a computer program to solve a specific problem as being comprised of three steps:

1. Conceptualising a solution to the problem in the domain of human cognition (the way a human would do it).
2. Symbolic translation of the human solution into computer operations (the way a computer could do it).
3. Concrete translation of the computational solution into a specific programming language (the generation of running code).

For example, if one wishes to implement a greedy algorithm for the Knapsack problem, the human solution would be expressed as "pick the highest value item that will fit"; the computer's solution would be something akin to "sort all items descending by value; iterate over the sorted list; compare each item's size to that remaining in the knapsack; choose the first one that fits". Note the much more decomposed and sequential nature of the computer's solution. Finally, the computer's solution would need to be written out in whatever programming language was being used, incorporating the particular syntactic rules and features of that language.

These three steps all involve both comprehension and production. For example, a programmer must be able to produce the computer's solution in step 2, and must be able to understand the logic of a computer's solution in order to translate it into code for step 3.

After identifying these three core activities of programming, we can then attempt to develop assessment tools that require these activities to be performed. In the next section, we present the technique we are currently exploring, and some preliminary evidence of its accuracy.

1.5 Assessment with Activity Diagrams

In recent offerings of our first programming course, we have begun using Activity Diagrams (e.g. Schmuller, 2004) as both a teaching and an assessment tool. Our preliminary data indicate that performance on these problems may correlate better with our Gold Standard (evaluation of artefacts) than do more traditional multiple-choice code reading questions.

Activity Diagrams are a component of the UML methodology for software development (e.g. Schmuller, 2004). They are used to diagram spatially the logical flow of a computer program, and have a notation for sequence, conditionals and looping. They do not depend on the syntactic details of any particular programming language. For our novice students, we use a simplified version of the full UML technique that eliminates some elements and requirements that are extraneous to the code written in a first programming course. An example for a program to play a simple "card game" is shown in Figure 1.

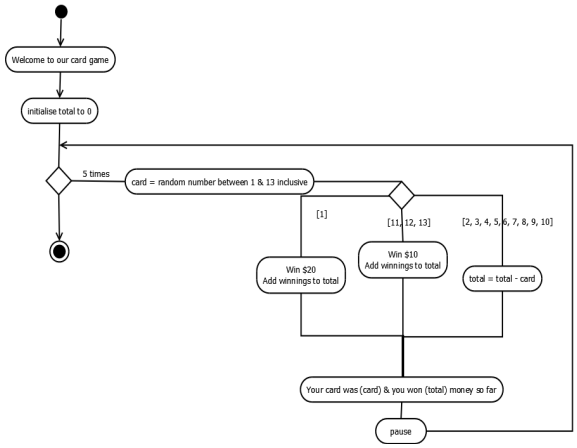

Figure 1: Activity Diagram for Simple Card Game Program

Activity Diagrams are similar to the traditional flowchart, which has been used to represent code structure for decades. In the early years of computer programming, the flowchart was studied extensively as a tool for program documentation, with mixed, sometimes controversial, results. After their introduction in the late 1940s (Goldstein and von Neumann, 1947) flowcharts rose rapidly in popularity to become an expected part of every programmer's skill set (cf. Schneiderman, Mayer, McKay et al., 1977). Flowcharts in these decades were used primarily as documentation tools. That is, a flowchart was produced to illustrate the structure and logic of a piece of software. Flowcharts were also used as an educational tool to illustrate complex algorithms (cf. Scanlan, 1987). Unfortunately, human factors studies performed in the 1980s demonstrated that flowcharts provided no advantage over an actual code listing for comprehension, debugging or modification (see e.g. Shneiderman, 1982). It was suggested that modern (for the time) high-level programming languages were more useful for representing program logic than were the graphical techniques of flowcharting (Ramsey, Atwood and Van Doren, 1983).

We use Activity Diagrams not for code documentation, but as teaching tools and for both formative and summative assessment. We have found them promising in all three contexts. However, in this discussion we will consider only their role in summative assessment or, equivalently, as a measure of programming ability in a research situation.

Previously, we described the journey from problem statement to working program as involving three steps: generating the human solution; translating to the computer's solution; translating into a specific language. Because Activity Diagrams are relatively language-agnostic, they give the student an opportunity to express the first two of these steps with a reduced burden of syntactic detail. By having the student translate an Activity Diagram into a specific programming language, we can test the third step.

In assessment, by selectively controlling the role of the activity diagram, we can isolate the three steps of digital artefact generation, and can assess both code production and comprehension. For example, if we give a student a problem statement in English and ask him or her to draw the corresponding Activity Diagram, we can observe the student's ability to produce the output of steps 1 and 2 (make a human solution; translate it into something the computer can do) from the procedure outlined above. If we give the student an Activity Diagram and ask him or her to explain the purpose of the resulting code, we are testing comprehension (code reading) of the same steps. Each of the various combinations of question content and task can serve to exercise one or more aspects of the programming process.

We believe that examination questions involving Activity Diagramming measure more accurately what real programmers do than the often abstract (and sometime artificial) code reading problems discussed above. In the present study, we wished to measure the relationships between student performance on a manually marked large programming project, assessment items involving Activity Diagramming, and assessment items using more traditional multiple-choice and short answer formats. Under the assumption that the project mark is the best available measure of true programming ability, we anticipated stronger correlations between project mark and scores on the Activity Diagram problems than between project mark and scores on the traditional problems.

2 METHODOLOGY

In two recent offerings of our CS1 first programming course, students' final course marks were computed as the weighted average of a) a set of in-class practical exercises; b) an in-class programming task under exam conditions; c) a large individual programming project and d) a written theory test performed under exam conditions. The in-class practicals were performed throughout the semester and the in-class programming task was performed half-way through the semester. These components were treated as both summative and formative assessment, with the students receiving detailed feedback on their progress, and additional tutorial support for any identified difficulties.

The out-of-class programming project was assigned four weeks before the end of the semester and was due on the last day. The written exam was given in the final week of the semester. These components were used for summative assessment.

The out-of-class programming project was a simple trivia game that required file I/O, random selection, and comparison. Upon submission, each student's solution was marked by an experienced programming teacher by hand using a detailed grading rubric[1], which assesses for both correct functionality and code quality. The marker checked carefully for instances of excess similarity between pairs of assignments, and no detectable incidences of plagiarism were identified. We acknowledge that stronger protection against plagiarism would be preferable (to insure, for example, that students are not getting help from more experienced programmers), but it is not practical to simultaneously provide an opportunity for a substantial programming exercise and to rigorously observe each moment of that programming process. For the present analysis, the marks on the out-of-class project serve as each student's Gold Standard. That is, the mark on a student's project is taken to be the most accurate reflection available of his or her true ability to produce a working digital artefact at the finish of this introductory programming course.

The written theory exam[2] contained 15 questions using a variety of formats including traditional multiple-choice code reading and writing questions, short code production exercises, and one Parsons Puzzle (Parsons and Haden, 2006). In addition, in one question (Question 12), students were given a syntactically correct code sample and asked to draw the corresponding Activity Diagram. In one question (Question 13), students were given a problem statement in English and asked to draw the corresponding Activity Diagram. All exams were marked by a single, experienced programming tutor.

72 students completed both the written theory exam and the out-of-class project, and their results are included in the following analyses.

3 RESULTS

Pearson-product moment correlations (point-biserial correlations for dichotomous problems) between the out-of-class project mark and question score were computed for each of the fifteen questions on the written theory exam. The results, along with the format of each exam question, are shown in Table 1.

[1 & 2] Available from Dale.Parsons@op.ac.nz

Question	Question type	Correlation with Project Mark	p-value
Q01	MCQ code writing	0.068	ns
Q02	MCQ code reading	0.149	ns
Q03	Short answer code reading	0.133	ns
Q04	MCQ code writing	0.252	p <.05
Q05	Short answer code reading	0.143	ns
Q06	MCQ code reading	0.216	ns
Q07	MCQ code reading	0.330	p<.01
Q08	Short answer code reading	0.183	ns
Q09	MCQ code reading	0.063	ns
Q10	Problem statement -> Code	0.224	marginal p=.059
Q11	Problem statement -> Code	0.423	p<.01
Q12	Code -> Activity Diagram	0.258	p<.05
Q13	Problem Statement -> Activity Diagram	0.352	p<.01
Q14	Short answer problem statement -> Code	0.279	p<.05
Q15	Parsons Puzzle	0.118	ns

Table 1: Correlations between exam questions and out-of-class project mark

Significant positive correlations with out-of-class project mark were found for two of the nine traditional MCQ or short answer questions (Questions 4 and 7). All three problems that required code writing were significantly (Questions 11 and 14) or marginally significantly (Question 10) correlated with project mark. Both Activity Diagram questions (Questions 12 and 13) were significantly correlated with project mark. All other observed correlations were non-significant.

The observed correlation between performance on the code writing questions (10, 11 & 14) and earned mark on the major code writing assignment is to be expected. One would assume that students who can write code well out of class are more likely to write code well on an exam. This correlation provides encouraging support for the validity of project mark as our "Gold Standard" of programming ability. It should be noted, however, that explicit code writing questions may be unable to discriminate between the three steps of the programming process proposed above. A student's failure to successfully write code on an exam might be due to difficulty translating his or her solution into computer operations, or due to difficulty expressing computer operations in syntactically correct programming language, or to some combination of the two. The complexity of explicit code writing exercises must also be severely restricted in the earliest weeks of a programming course, where syntactic mastery has not been achieved.

Among the best predictors of out-of-class project mark were the two questions where students were given a code sample or a problem statement and asked to generate an Activity Diagram.

This type of problem maps directly to our posited process for generating working code. It captures a student's ability to perform critical computational thinking, with a reduced burden on syntactic accuracy. It tests this aspect of their ability to program in a way that is neither artificially complex nor artificially abstract. And, at least for this sample, it correlates significantly with the ability to produce a nontrivial working application (i.e. the ability to program).

While it is of interest to note that performance on the Activity Diagram questions is significantly correlated with the score on the out-of-class project, it is equally interesting to note that performance on the majority of the traditional format questions is not. For example, question 2 is a standard multiple-choice code reading question:

What are the values of girls, boys, and children after the following code has been executed?

```
int girls = 0;
int boys = 0;
int children = 0;
children = girls + boys;
girls = 15;
boys = 12;
```

(a) 0, 0, 0
(b) 0, 0, 27
(c) 15, 12, 0
(d) 15, 12, 27

The observed correlation between marks on Question 2 and our Gold Standard measure of programming ability is small -- only 0.149 -- and nonsignificant.

Non-significant correlations between performance on a test item and mark on the out-of-class project can be caused by a variety of numerical patterns. Most obviously, a low correlation occurs when success on the test item is not consistently associated with a high mark on the project, and symmetrically, failure on the test item is not consistently associated with a low mark on the project. However, a low correlation will also occur if the test item is so easy that most students get it right (ceiling effect) or so difficult that most students get it wrong (floor effect). In this case, performance on the item is largely the same for all students regardless of performance on the project, and the Pearson-product moment correlation will be near zero. In either of these scenarios, score on the test item is not a sensitive measure of programming ability.

A summary of performance for each exam question is shown in Table 2. The nine traditional format questions (Q1 to Q9) are graded "all or nothing", that is, no partial credit is given. For those items, Table 2 presents the percent of correct responses across all students. Note that the multiple-choice questions (numbers 1, 2, 4, 6, 7 & 9) each provide four response alternatives, so one would expect a 25% correct response rate simply due to chance. Problems 10 to 15 are marked out of a fixed number of points, and partial credit is given. For those problems, Table 2 shows the average proportion of available marks earned, across all students.

Question	Question type	Percent Correct	Average Pr(Marks)
Q01	MCQ code writing	0.81	
Q02	MCQ code reading	0.78	
Q03	Short answer code reading	0.93	
Q04	MCQ code writing	0.81	
Q05	Short answer code reading	0.71	
Q06	MCQ code reading	0.60	
Q07	MCQ code reading	0.47	
Q08	Short answer code reading	0.71	
Q09	MCQ code reading	0.86	
Q10	Problem statement -> Code		.76
Q11	Problem statement -> Code		.66
Q12	Code -> Activity Diagram		.79
Q13	Problem Statement -> Activity Diagram		.40
Q14	Short answer problem statement -> Code		.59
Q15	Parsons Puzzle		.86

Table 2: Performance summary for all exam questions.

The values in Table 2 suggest that both sources of low correlation described above are present in our results. Question 3, for example, shows a 93% correct response rate and a non-significant correlation of 0.133 with the out-of-class project. Since most students answered this question correctly, it fails to discriminate between those students who perform well on the out-of-class project and those who do not. Question 6, in contrast, shows a moderate 60% correct response rate and a non-significant correlation of 0.216. This question does not appear to suffer from a floor or ceiling effect, so performance on it is simply not well-correlated with assignment mark. In both cases, these traditional format questions are not usefully predictive of performance on the out-of-class project.

Our Questions 1 to 9 are representative of the type often used to assess programming ability in computer science education research, including those studies which conclude that our students cannot program because of low success rates on these test items. If in fact, the ability to solve this kind of test item is not well correlated with the ability to program, we may be encouraged to posit that our students may, in fact, be able to program, but that this type of question does not accurately measure that ability.

These results are of course only preliminary. The sample size is only moderate, and one should not infer too much from individual, possibly pathological, exam questions. Nonetheless, there is a clearly detectable pattern that encourages us to believe that through the use of Activity Diagram questions, we might be able to obtain an acceptably accurate measure of true programming ability without the often prohibitive cost of hand-marking complete software artefacts.

4 DISCUSSION

In recent decades, computer programming has progressed from an isolated esoteric tool, to a critical element of a scientific discipline, to a vocational practice on which many of the world's systems and institutions now depend. Throughout this process, programming pedagogy has struggled to achieve an always successful delivery of what is an inherently difficult subject to teach. This difficulty has led to the gloomy conclusions that we can't teach it, and our students can't do it.

In this paper, we have suggested that this sad situation may be due, at least in part, to the methods we use to assess programming ability, both in research and in the classroom. We propose that the types of questions often used to assess programming ability are not measuring its most essential aspect – the ability to produce good quality working code. As an alternative, we propose the use of a spatial representation of coding logic – the Activity Diagram – that may more accurately reflect what we mean when we ask "can our students program"? Our early experience with this technique indicates that it correlates better with the ability to produce quality working code than do traditional multiple choice and short answer questions.

One of the reasons that our students score so poorly on traditional written programming questions is that they are often confusing. In order to avoid trivially simple non-discriminatory assessments, we must make these questions convoluted and abstract, often to the point where they no longer represent the realistic cognitive behaviours of real programming. Activity Diagram questions, in comparison, are inherently complex. To generate an Activity Diagram, the student must generate a computationally appropriate solution to a problem, and express that solution using the Activity Diagram notation. There is enough challenge in this activity that we do not need to artificially complicate the problem to avoid triviality. We can use realistic programming contexts to assess real programming skill.

Our results, while preliminary, encourage us to hope that perhaps we haven't really been doing such a bad job of teaching programming, although we may have been doing a questionable job of assessing it.

5 REFERENCES

Bergin, S. and Reilly, R. (2005): The influence of motivation and comfort level on learning to program. *Proceedings of the 17th Annual Workshop on the Psychology of Programming Interest Group* pp 293-304, University of Sussex, Brighton UK 29, June – 1 July, 2005.

Bornat, R., Dehnadi, S., and Simon (2008): Mental models, consistency and programming aptitude. *ACE '08: Proceedings of the tenth conference on Australasian computing education* Vol. 78.

Cardell-Oliver, R. (2011): How can software metrics help novice programmers? *ACE 2011, Proceedings of the 14th Australasian conference on Computing education*, Perth.

Clear, T. (2008): Thinking issues: assessment in computing education: measuring performance or conformance? *SIGCSE Bulletin* 01/2008; 40:13-15.

Dehnadi., S. (2006): Testing programming aptitude. *Proceedings of the Psychology of Programming Interest Group* 18th Annual Workshop, 22-37.

Fincher, S. (1999): What Are We Doing When We Teach Programming? *29th ASEE/IEEE Frontiers in Education Conference* San Juan, Puerto Rico

Ford, M. and Venema, S. (2010): Assessing the Success of an Introductory Programming Course. *Journal of Information Technology Education* 9:133-145.

Goldstein, H.H., and von Neumann, J. (1947): Planning and coding problems for an electronic computing instrument, part II, vol I. Report prepared for the U.S. Army Ordinance Dept. Reprinted in von Neumann, J. *Collected Works, Vol. V, A.H.* Taub, Ed., Mc-Millan, New York, pp. 80-151.

Gonzalez, G. (2006): A systematic approach to active and cooperative learning in CS1 and its effects on CS2. *SIGCSE 2006*, March 1-5, 2006, Houston, TX, USA.

Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva and Tadeusz Wilusz (2013): A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations *ITiCSE-WGR'13* June 29-July 3, 2013, Canterbury, England, UK.

Ihantola, P., Ahoniemi, T., Karavirta , V. and Seppälä, O (2010): Review of Recent Systems for Automatic Assessment of Programming Assignments. *Koli Calling '10*, October 28-31, 2010, Koli, Finland

Lahtinen, E., Ala-Mutka, K. and Järvinen, H-M. (2005): A study of difficulties of novice programmers. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Monte de Caparica, Portugal, June 27-29, 2005.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J. , Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. (2004): A multi-national study of reading and tracing skills in novice programmers, *ACM SIGCSE Bulletin*, 36(4).

Lister, R., Simon, B., Thompson, E., Whalley, J.L. and Prasad, C. (2006): Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ITICSE '06 Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*. 118-122

Ma, L., Ferguson, J., Roper, M., and Wood, M. (2007): Investigating the viability of mental models held by novice programmers. *Proceedings of the 38th SIGCSE technical symposium on Computer science education* (SIGCSE '07). ACM, New York, NY, USA, 499-503.

McCartney, R., Boustedt,J., Eckerdal, A., Sanders, K., and Zander, C. (2013): Can First-year Students Program Yet? A Study Revisited *ICER'13*, August 12–14, 2013, San Diego, California, USA.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001): A multi- national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-140.

Parsons, D and Haden, P. (2006): Parson's programming puzzles: a fun and effective learning tool for first programming courses, *Proceedings of the 8th Australasian Conference on Computing Education*, p.157-163, January 16-19, 2006, Hobart, Australia.

Pears, A., Seidman, S., Malmi, L., Mannila, L. Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007): A survey of literature on the teaching of introductory programming. *SIGCSE Bulletin*, 39(4), 2007.

Ramsey, H.R., Atwood, M.E. and Van Doren, J.R. (1983): Flowcharts Versus Program Design Languages: An Experimental Comparison. *Communications of the ACM*, 26(6):445-449.

Sajaniemi, J., Kuittinen, M. and Tikansalo, T. (2008): A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course. *ACM Journal on Educational Resources in Computing*, 7(4).

Scanlan, D. (1987): Data-Structure Students May Prefer to Learn Algorithms Using Graphical Methods. SIGCSE '87 *Proceedings of the eighteenth SIGCSE technical symposium on Computer science education*. pp. 302-307.

Schmuller, J. *Sam's Teach Yourself UML in 24 Hours, Complete Starter Kit* (3rd Edition) Sams Publishing, 2004.

Sheard, J. (2012): Exams in computer programming: What do they examine and how complex are they? In *Proceedings of the 23rd Annual Conference of the Australasian Association for Engineering Education*.

Shneiderman, B., Mayer, R., McKay, D. and Heller, P. (1977): Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373-381.

Shneiderman, B. (1982): Control Flow and Data Structure Documentation: Two Experiments. *Communications of the ACM*, 25(1):55-63.

Simon, Sheard, J. Carbone, A., Chinn, D., Laakso, M., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, A. and Warburton, G. (2012): Introductory programming: examining the exams *Proceedings of the Fourteenth Australasian Computing Education Conference*, Melbourne, Australia

Thomas, L., Ratcliffe, M., Woodbury, J., Jarman, E. (2002): Learning styles and performance in the introductory programming sequence . *SIGCSE '02 Proceedings of the 33rd SIGCSE technical symposium on Computer science education.*

Vamplew , P., and Dermoudy, J. (2005): An anti-plagiarism editor for software development courses *ACE '05 Proceedings of the 7th Australasian conference on Computing education* 42:83-90.

Whalley , J., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K. and Prasad, C. (2006): An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies *ACE '06 Proceedings of the 8th Australasian Conference on Computing Education.* 52:243-252.

Wing, J. (2006): Computational Thinking *Communications of the ACM*, 49(3):33-35.

Robins, A. (2010): Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, 20:37 - 71.