Relationships between Logic Depiction, UML Diagramming and Programming

Amitrajit Sarkar
Christchurch Polytechnic Institute of Technology
130 Madras Street,
Christchurch
+64 3 940-8000
amitrajit.sarkar@cpit.ac.nz

Rob Oliver
Christchurch Polytechnic Institute of Technology
130 Madras Street,
Christchurch
+64 3 940-8000
rob.oliver@cpit.ac.nz

Mike Lopez
Christchurch Polytechnic Institute of Technology
130 Madras Street,
Christchurch
+64 3 940-8000
mike.lopez@cpit.ac.nz

Mike Lance
Christchurch Polytechnic Institute of Technology
130 Madras Street,
Christchurch
+64 3 940-8000
mike.lance@cpit.ac.nz

ABSTRACT
Beginning programmers are often taught to design algorithms in pseudo code, a structured form of English, before implementing the algorithms in code. This approach is often advocated because it is seen as enabling programmers, and especially novice programmers, to reason about program logic without the distraction of the specific syntax of a programming language, and because it can be used as a basis for program documentation. Similar arguments are often given for the use of UML diagrams. In recent semesters, we have trialled the programming language Scratch as an alternative to structured English for pseudo code. This paper uses assessment data to investigate the relationship between pseudo code (both structured English and Scratch programs), UML, and programming ability. We found a consistent and strong relationship between programming and UML diagramming skills, but a relatively weak relationship between programming and either form of pseudo code. These findings lead us to question the value of teaching pseudo code and our motives for teaching it.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms
Algorithms, Documentation, Design, Standardization, Languages.

Keywords
Novice programmers, introductory programming, pseudo code, UML.

1. INTRODUCTION
The ability to conceptualise and plan the solution to any programming problem (more commonly described as an algorithm), whether that problem is simple or complex, lies in the ability of the programmer to be able to produce a series of stepwise progressions through the problem from start to finish. The requirement to validate that solution (desk check), suggests the use of a programming language independent syntax to describe each step of the proposed solution. Pseudo code (which is a form of structured English having no internationally recognised standards) is such a language. It uses a structured set of statements that describe each individual step of logic while retaining programming language independence. This allows the developer to plan logical solutions, and verify that they will work, without requiring knowledge of the syntax of any specific programming language. Thus, errors in logic can be identified before the costly process of program coding commences. Taken to its utmost level, pseudo code allows the developer to plan the structure of the solution down to a program instruction level. A possible representation in pseudo code of the problem of calculating the area of a circle is shown in Figure 1.

```
calculateCircleArea
        call initializeAndGetData
        call calculateArea
        call showResults
initializeAndGetData
        set circleArea to 0
        set circleRadius to 0
        set temporaryArea to 0
        show "Enter the circle radius"
        obtain circleRadius
calculateArea
        square circleRadius giving temporaryArea
        multiply temporaryArea by 3.1416 giving circleArea
showResults
        show "Circle area is " , circleArea
        show "All processing complete"
```

Figure 1: Pseudo code to calculate the area of a circle

This quality assured paper appeared at the 3rd annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2012) incorporating the *25th Annual Conference of the National Advisory Committee on Computing Qualifications*, Christchurch, New Zealand, October 8-10, 2012. Mike Lopez and Michael Verhaart, (Eds).

A possible implementation of this in the Scratch programming language (Resnick, et al., 2009) is shown in Figure 2.

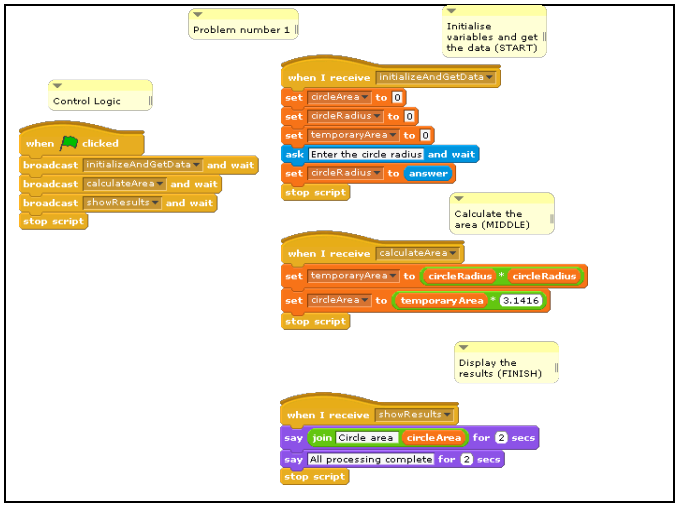

Figure 2: Calculating the area of a circle in Scratch

Historically, pseudo code was regarded either as informal textual representations of a program or algorithm, or in an extended sense, as also including all of the annotations and sketches made by programmers, whether textual or not (Bellamy, 1994). In general, these were informal, idiosyncratic personal formalisms that represented the programmer's abstract computational model of a problem solution. The most common shared features of these were: drawing elements of a solution in boxes, drawing arrows between boxes to represent relationships, and adding annotations for clarification and elaboration. At that time, the broad purpose of pseudo code was to help a programmer conceptualise a problem and its solution and diagrams generally provide greater cognitive support for conceptualisation than pure text representations (Larkin & Simon, 1987).

There have been some attempts to standardise and formalise both the textual representations and the sketches and annotations, thus allowing these to form the basis of communication to others. In his *Pseudo-code Programming Process*, McConnell (1993) recommends using pseudo code as an intrinsic part of the program design and construction process, both during the development of the design and as the comments in the ultimate program. There have also been some attempts to formalise the pseudo code itself (Roy, 2006), and some educators have defined their own standards for their courses; for an example see Dalbey (2012). However, no consensus has been achieved on a standard, nor even that a standard is desirable.

Although no standard formalism of pseudo code has yet emerged, it is most commonly expressed as structured English in which a separate line is used for each step in a sequence, and control structures similar to those used in programming languages are introduced. These control structures typically follow the author's preferred programming language and comprise statements such as IF … THEN … ELSE … END IF or WHILE … END WHILE.

In contrast, formalising the sketches and annotations gave rise to the Unified Modelling Language (UML) as a means of representing and defining the characteristics of a solution. UML is a graphical language for visualizing, specifying, constructing and documenting the artefacts of a software-intensive system. For many, UML is much more than that and symbolises the transition from code-oriented to model-oriented software production techniques. Modelling is an essential part of large software projects, and helpful for medium and even small projects as well. In the early 1990s there was a so-called "method war" between the competing object-oriented methodologies of Booch (1991), Jacobson, Christerson, Jonsson, and Overgaard (1992) and Rumbaugh, Blaha, Premerlani, Eddy, and Lorenson (1991). These authors joined forces and combined their most useful concepts into the UML (Jacobson, Booch, & Rumbaugh, 1999). Publication of UML standards in mid-1990s by the Object Management Group (OMG) not only standardised the notation for Object Oriented analysis and design but also effectively put an end to this method war and UML is now regarded as a de-facto standard for modelling Object Oriented systems.

Nowadays, although experienced programmers have embraced UML, few write pseudo code before coding. As Marrer notes:

> As you become a more experienced programmer you might find yourself moving away from a pseudo code model. I think, if asked, most experienced programmers would tell you that they do not use pseudo code unless it is required … (2009, p. 350)

This is understandable because the essential idea underpinning the advocacy of pseudo code is that it is easier to read and understand an algorithm expressed in structured English, than its implementation in code. Although this may well be true for beginning programmers, it is clearly doubtful for experienced programmers. Indeed, most will find a programming language clearer and more expressive than English for this purpose. From this perspective, we believe the main role of pseudo code in teaching is to support a novice programmer in the early stages as they build expertise, and acquire design patterns, on their journey to becoming more expert.

Although pseudo code is sometimes advocated as a source of comments in programs (McConnell, 1993), the main purpose of comments is to help a reader understand the code and pseudo code is not ideal for this purpose. What experienced programmers like is for comments to express the intent of the author of the code. As Kernighan & Plauger note in their classic work (1974), "Comments should provide additional information that is not readily obtainable from the code itself. They should never parrot the code." (p. 152). Useful comments tell us not what the code does – self-evident in well written code – but why it has been written that way. There is also a downside to formalism. The effort required to create formal documentation may cause premature shaping of a solution, leading to insufficient exploration of other design possibilities. Although standardisation facilitates communication to others, there is thus still a need for informal techniques that support the process of developing a conceptualisation of a system. Indeed, a quick look at a whiteboard in any programmer's office may well reveal diagrams looking very similar to those of Bellamy's (1994) study.

In sum, although there is good justification for teaching UML diagramming, we had some doubts about the need to teach pseudo code. These doubts about the relevance of pseudo code in the modern era led us to introduce the programming language Scratch in our introductory programming course as an alternative method of logic depiction to pseudo code. We believed that not only would this support novice programmers in their early development of algorithms, but that it would also provide an engaging start to their journey of learning to program. Our goal in this study was to use objective evidence to investigate the relationships among pseudo code, UML diagramming, and programming performance.

2. SAMPLE

BCSE101 at Christchurch Polytechnic Institute of Technology (CPIT) is comparable to CS1 at many institutions. It is a compulsory introductory level programing course taught to first year students in their first semester of study. The intention of this course is to introduce sufficient analysis and design techniques to introduce students to object oriented programming. The course is a combination of analysis and design theory taught from the UML and Rational Unified Process (RUP) perspective and an introduction to software development taught in a teaching language rather than a commercial language. This course is taught as a pure introduction to software development in a language which encourages, and indeed forces, the student to use good programming practices. Objects first is emphasised and the problem domains mirror those modelled in UML. Currently it is taught in Jade for most of the semester with SCRATCH used for the first five weeks. The main reason behind this is to expose the beginner level students to a relatively syntax-free coding environment in the early stages so as to make them seriously affianced with the subject by lowering the barrier to code.

Our analysis is based on two datasets. The first dataset (n=165) comprised the assessment results for two semesters of 2009 and 2010 in which pseudo code was used as the method of logic depiction. The second dataset (n=56) comprised the assessment results for second semester of 2011 in which the Scratch programming language was used for logic depiction. We introduced Scratch in semester one of 2011 but deliberately skipped that data in this analysis to avoid any artefacts associated with the first delivery.

Both datasets had the same assessment structure. There were six assessments in total. The first assessment was a pseudo code test. The second was a class diagram assignment which comprised identifying different classes and drawing a static analysis level class diagram. The third assessment was a programming assignment where students implement part of the class diagram. The fourth assessment was a System development assignment where students implement a fairly complex system based on a set of business rules. In this assignment they created new classes, wrote new tests and produced outputs. The penultimate assessment was an in-class practical test and the final assessment was a multi-choice theory exam where questions were asked on UML structure, behaviour and interaction diagrams. This assessment had a strong relationship with the system development assignment.

3. METHOD

We mapped the six assessments in the course to three scales: logic depiction, UML diagramming and programming. The logic depiction scale comprised the single assessment for pseudo code or Scratch, respectively. The UML diagramming scale comprised two assessments. The first was the class diagram assignment and the second was the final examination. The programming scale comprised the remaining three programming assessments: two assignments and a practical test.

The scales were acceptably homogeneous. For a composite scale comprising all assessment activities, homogeneity (Loevinger, 1948) was H=0.660 for the pseudo code dataset and H=0.673 for the scratch dataset. With these coefficients, the scales are classified as *strong* scales under the criteria of Mokken and Lewis (1982, p. 422). The corresponding figures for Cronbach's alpha were 0.888 for the pseudo code dataset and 0.868 for the scratch dataset. We used a Rasch model (Andrich, 1978; Rasch, 1960/1980) to build the scales. This allowed us to create interval level measures from the ordinal assessment scores and carry out a number of diagnostics on data quality. Under the guidelines given by Wright and Linacre (1994), all scales were classified as *productive* with no *degrading* items. Some local dependence was diagnosed among the items. However, this only affects estimated standard errors for person measurements and these are not pertinent to this study.

Having established interval level measurement for the main study variables, our approach to subsequent analysis was correlational. One threat to validity in our approach relates to the logic depiction assessment (pseudo code or scratch). This assessment was a mandatory test, and students who scored less than 50% on their first attempt were given the opportunity to retake the test after further study. Only the final outcome was recorded in the assessment results. From an analysis perspective, the main effect of this is that the variability of the logic scale had a restricted range.

4. RESULTS

The first order correlations between the study variables in the pseudo code dataset are shown in Table 1.

Table 1: First order correlations between variables

Variable	UML	Programming
Pseudo code	0.306	0.362
UML		0.804

From this table it can be seen that all variables are positively inter-correlated. Disentangling these relationships requires finding the unique contributions of the variables after controlling for the influence of the other variables. We carried out a path analysis to determine these unique contributions. The resulting path diagram for the pseudo code dataset is shown in Figure 3.

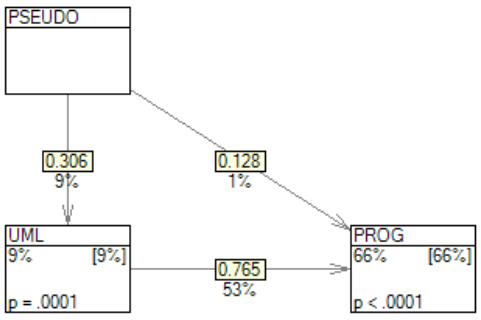

Figure 3: path diagram for pseudo code dataset

In this figure, the variables are shown in titled boxes. For predictor variables, the contents of the box are empty. For criterion variables, the box shows the percentage of variance explained (R^2), this percentage adjusted for the number of predictors shown in square brackets (Theil, 1961), and the probability of observing a relationship this strong in a random sample. Arrows are drawn from predictors to criterion variables. The box on the arrow shows the Beta weight of the path, and the squared semi-partial correlation is shown under the box. This last

indicates the unique contribution of the variable to the overall variance explained.

There is a strong relationship between performance in UML diagramming and programming. This relationship accounts for 53% of the variability in programming performance. The unique contribution of pseudo code is relatively small (1%) and adds little to the explanation of UML alone; UML as a sole predictor explains 65% of the variability of programming in this dataset. The relationship between pseudo code and UML diagramming is much weaker, accounting for 9% of the variability in UML diagramming performance.

The 12% of explained variability in programming performance remaining after the unique contributions of pseudo code and UML have been accounted for represents commonality between these two predictors. This commonality has two components: structural and personal factors. The structural factor relates to any intrinsic relationship between pseudo code and UML and the contribution this makes to programming performance. Personal factors relate to the natural variability among students; some students may be more attuned to programming, more motivated to succeed, pay closer attention to marking criteria, apply greater effort, have greater ability, and so on. It is not possible to disambiguate these two factors within the present study design. However, given that one would expect significant personal factors to be present in any educational dataset, the magnitude of this commonality suggests that, after accounting for personal factors, the residual intrinsic structural relationship between pseudo code and UML diagramming, as measured by the effect on programming performance, is relatively small.

For comparison, the path diagram for the Scratch dataset is shown in Figure 4.

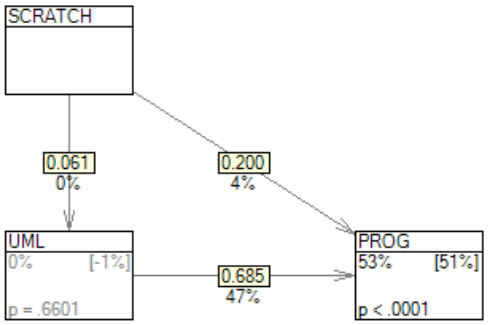

Figure 4: Path diagram for the Scratch dataset

The overall pattern is very similar in this Scratch dataset. UML performance is again a strong predictor of performance in programming, accounting for 47% of the variability. The unique contribution of Scratch performance is again relatively small (4%) and adds little to the explanation of UML alone; UML as a sole predictor explains 49% of the variability of programming in this dataset. In contrast to the pseudo code dataset, performance in Scratch as a logic depiction method has no significant association with the performance of UML diagramming performance.

5. DISCUSSION

Both datasets showed a strong relationship between performances in UML diagramming and programming. Performance in pseudo code added little to the explanation of programming performance over that explained by UML performance. Likewise, although slightly stronger, performance in Scratch, added little to the explanation of programming performance given by UML performance. For performance in UML diagramming, the relationship with pseudo code performance was relatively weak and the relationship with programming in Scratch was not significant.

Initially, we were surprised by the relatively small connection between programming performance, and performances in the logic depiction methods of pseudo code and Scratch. However, on reflection, one explanation may lie in the difficulties students have long had with introductory programming courses. For example, Bergin and Reilly note that "It is well known in the Computer Science Education (CSE) community that students have difficulty with programming courses and this can result in high drop-out and failure rates" (2005, p. 293). From a student's perspective, rather than making algorithm construction easier, as we intend, logic depiction methods might be just one more thing that the student has to learn.

As noted earlier, the fact that students who failed the logic depiction assessment on their first attempt were allowed to re-take the assessment after further study poses a potential threat to validity because of reduced variance. However, this affected relatively few students and cannot, on its own, explain the low association. Another potential threat to the validity of these findings is that the timing of the logic depiction assessment might have been the cause of a weak relationship. Perhaps changes happened in student understanding after the students completed the pseudo code assignment, or, perhaps, early assessments are simply not a good predictor of later performance? However, the strong coherence of the scales suggests there is little evidence of either of these.

The evidence of the assessment data thus strongly point to a low intrinsic association between logic depiction methods and programming after controlling for UML diagramming performance. There are two ways this low association can be interpreted. First, logic depiction methods might not be of much help to a student in the construction of algorithms in code. Second, success in programming might simply require a great deal more than just algorithm construction. In either case, the findings suggest that our present emphasis on logic depiction methods over and above UML diagramming might be misplaced. In summary, proficiency in UML diagramming seems to be a sufficient non-programming method of reasoning about a program.

6. CONCLUSION

These findings lead us to question our motives for teaching pseudo code in any form. Does it really help students' learning? Does it help them become a better programmer? Or, maybe, we just teach it because everyone else does?

7. REFERENCES

Andrich, D. (1978). A rating formulation for ordered response categories. *Psychometrika, 43*, 561-73.

Bellamy, R. (1994). What does pseudo-code do? A psychological analysis of the use of pseudo-code by experienced programmers. *Human-Computer Interaction*, 225-246.

Bergin, S., & Reilly, R. (2005). The influence of motivation and comfort-level on learning to program. *In Proceedings of the 17th annual workshop of the Psychology of Programming Interest Group* (pp. 293-304). Brighton, UK: University of Sussex.

Booch, G. (1991). *Object-Oriented Design with Applications.* Redwood City, CA: Benjamin Cummins.

Dalbey, J. (2012, 8 18). *Pseudocode standard.* Retrieved from http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process.* Reading, MA: Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach.* Redwood City, CA: Addison-Wesley.

Kernighan, B., & Plauger, P. (1974). *The elements of programming style.* New York: McGraw-Hill.

Larkin, J., & Simon, H. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11*, 65-99.

Loevinger, J. (1948). The technique of homogeneous tests compared with some aspects of "scale analysis" and factor analysis. *Psychological Bulletin, 45*, 507-530.

Marrer, G. (2009). *Fundamentals of programming with object oriented programming (Python edition).* Glendale, AZ: Laptop Press.

McConnell, S. (1993). *Code complete: A practical handbook of software construction.* Redmond, Wa: Microsoft Press.

Mokken, R., & Lewis, C. (1982). A nonparametric approach to the analysis of dichotomous item responses. *Applied Psychological Measurement, 6*(4), 417-430.

Rasch, G. (1960/1980). *Probabilistic models for some intelligence and attainment tests. (Copenhagen, Danish Institute for Educational Research), expanded edition (1980) with foreword and afterword by B.D. Wright.* Chicago: The University of Chicago Press.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60-67.

Roy, G. G. (2006). Designing and explaining programs with a literate pseudocode. *Journal on Educational Resources in Computing (JERIC), 6*(1), 1-18.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorenson, W. (1991). *Object-Oriented Modelling and Design.* New York: Prentice Hall.

Theil, H. (1961). *Economic forecasts and policy.* Amsterdam: North-Holland Pub. Co.

Wright, B., & Linacre, J. (1994). Reasonable mean-square fit values. *Rasch Measurement Transactions, 8*(3), 370.