An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies

Jacqueline L. Whalley
Computer and Information Sciences
Auckland University of Technology
Private Bag 92006, Auckland 1020,
New Zealand
jacqueline.whalley@aut.ac.nz

Raymond Lister
Faculty of Information Technology
University of Technology, Sydney
Broadway, NSW 2007,
Australia
raymond@it.uts.ac.au

Errol Thompson
Department of Information Systems
Massey University
Wellington
New Zealand
E.L.Thompson@massey.ac.nz

Tony Clear and Phil Robbins
Computer and Information Sciences
Auckland University of Technology
Private Bag 92006, Auckland 1020,
New Zealand
tony.clear@aut.ac.nz
phil.robbins@aut.ac.nz

P. K. Ajith Kumar
Bay of Plenty Polytechnic,
Tauranga,
New Zealand
Ajith.Kumar@boppoly.ac.nz

Christine Prasad
School of Computing and
Information Technology
Unitec New Zealand
Auckland,
New Zealand
cprasad@unitec.ac.nz

Abstract

In this paper we report on a multi-institutional investigation into the reading and comprehension skills of novice programmers. This work extends previous studies (Lister 2004, McCracken 2001) by developing a question set within two key pedagogical frameworks: the Bloom and SOLO taxonomies. From this framework of analysis some interesting emergent patterns relating the cognitive level of the questions to student performance have been identified.

Keywords: Bloom's Taxonomy, SOLO, novice programming, multi-institutional.

1 Introduction

A 2001 ITiCSE working group assessed the programming ability of an international population of students from several universities (McCracken et. al. 2001). The students were tested on a common set of program-writing problems and the majority of students performed more poorly than expected. It was not clear why the students struggled to write the required programs. One possible explanation is that students lacked knowledge of fundamental programming constructs. Another possible explanation is that students were familiar with the constructs but lacked the ability to "problem solve".

In 2004, another ITiCSE working group (the "Leeds group") attempted to investigate some of the reasons why students find programming difficult (Lister et. al. 2004). The working group attempted to benchmark the program-reading skills of novice programmers. They found that many students could not answer program reading problems, "suggesting that such students have a fragile grasp of skills that are a pre-requisite for problem-solving".

As interesting as the results are from the Leeds group, the project did not have a sufficient theoretical underpinning. The choice of the reading problems was not informed by a theoretical model. The multiple choice problems were taken from past exam papers written by Lister. The only criterion for choosing the problems was the percentage of students who had answered each question correctly. Furthermore, the analysis of the data was not driven by any theory or model of how students should solve or actually address problems.

The work described in this paper uses a revised version of Bloom's taxonomy (Anderson et al. 2001) to generate and analyse program reading questions and uses the SOLO taxonomy (Biggs and Collis 1982) in the analysis of some of the data.

1.1 Project Timeline and Organisation

The authors are based at five different tertiary education institutions. They first met to discuss this project in December 2004. At that meeting, a draft instrument was developed, using the revised Bloom's taxonomy.

For the next 6 months, authors at three of the participating institutions collected data from their students. In July 2005, the authors met again to analyse and discuss their results. It was at this meeting that the SOLO taxonomy was introduced as a tool for analysing the data.

2 Question-writing and Bloom's Taxonomy

The Leeds group divided their multiple choice questions into two broad types. The first type of question was a 'fixed-code' question. In questions of this type, students were given a short piece of code and asked to determine the value output, or the value contained by a variable, after that code was executed. The second type of multiple choice question was a 'skeleton-code' question. In this type of question, students were given a piece of code with one or more lines missing. They were also told the intended function of the code, and were asked to select from the available options the code that would correctly complete the skeleton. The Leeds group observed that, as a general rule, students found 'fixed-code' questions easier than 'skeleton-code' questions, but they did not offer an explanation of why that should be the case.

'Fixed-code' questions may be considered to fit into the *Executing* subcategory of the cognitive process category *Apply*. This is described in the revised Bloom's taxonomy as "applying a procedure to a familiar task".

Categorizing 'skeleton-code' questions into the revised Bloom's taxonomy is more difficult than fixed-code questions, because students appear to manifest a greater variety of approaches to solving 'skeleton-code' questions. A student might solve a 'skeleton-code' multiple choice question via the cognitive process of *Apply*, in the subcategory of *Executing*. That is, the student might substitute each of the given options into the skeleton, execute the code for each of those substitutions, and then select the substitution that led to the intended behaviour. On the other hand, a student might solve a skeleton-code multiple choice question by reasoning about how the parts of the code relate to one another. Depending upon exactly how that was done, the student could be operating at either the *Analyse* or *Evaluate* levels. In some rare instances a student may even be operating at the *Create* level.

The revised Bloom's taxonomy is a fertile source of ideas for question generation. After examining the categories in the revised taxonomy, and relating them to programming, we found that more question types than just 'fixed-code' and 'skeleton-code' questions may be generated, as is illustrated in the next section of the paper.

3 Study Instrument Development

The set of 10 questions surveyed in this paper, consisted of 9 multiple choice questions and one short-answer question. In order to identify what was being tested in terms of programming ability the working group decided to develop an instrument that was built upon a framework. Two of the multiple choice questions used in this study (1 and 9) were taken directly from the Leeds working group instrument (5 and 2 respectively; Lister 2004). These two questions allowed for some direct comparative analysis.

The remaining 8 questions were designed by the working group, using the revised Bloom's taxonomy, with the express purpose of devising a diverse set of questions to evaluate the program comprehension skills of novice programmers.

The complete set of 10 questions is available from a web site (Bracelet). To conserve space in this paper, the 10 questions are summarised as follows:

Q1: A "fixed-code" question, identical to Question 5 from the Leeds working group. The code contains a single "while" condition, with a simple terminating condition that does not involve conjunctions or disjunctions, with five assignment statements within the loop.

Q2: Students were given a piece of code and were required to find the matching flow chart.

Q3: Similar to question 2, but in this case the students were supplied with a structure diagram and had to select the piece of code that performed the same task.

Q4. Students were given a piece of code, a single "if" statement with an "else" component, that set a boolean variable to true if an integer variable contained a value within a given range; otherwise the boolean variable was set to false. The students were required to identify a functionally equivalent piece of code from the options. Each option consisted of an "if/else" statement. Each "if" statement's boolean condition contained either a conjunction or a disjunction.

Q5: A "fixed-code" question. The code consisted of nested "for" loops, with a simple "if" condition within the inner loop.

Q6: The code contained a single "while" condition, which stepped an integer variable "i" through a series of values. The code also contained a single array initialized to a set of values. The student was required to choose from among the four options the best English language description of what the code did.

Q7: The complete text of this question is given later in the paper. The students were given some buggy code, told the intended function of that code, and given an example of the incorrect output of the buggy code. The students were then asked to select a change to the code that would fix the bug.

Q8: A "skeleton-code" question, with part of one line missing, the boolean condition of an "if" statement. The code was intended to check whether the characters stored in an array formed a palindrome. The code contained a single "while" statement, with a boolean condition containing two conjuncts. The body of the loop consisted of two assignment statements. The missing "if" conditions immediately followed the "while" loop. Students had to choose the appropriate boolean condition for the missing 'if' statement.

Q9: A "fixed-code" question, identical to question 2 from the Leeds working group. The code contained a single "while" condition, with a complicated "if" condition within it. The students were only given the code. They were not told the function of the code, which was to count the number of identical elements in two sorted integer arrays omitting the first element in both arrays.

Q10: The complete text of this question is given later in the paper. This is the only question in the complete set of questions which is not a multiple choice question. Instead, students were given a short piece of code and asked to describe the purpose of the code "in plain English".

The instrument also contained an 11th question which is not discussed in this paper.

3.1 Bloom Categorisation, Version 1

Given our aim of investigating how novices comprehend code, the goal was to design a set that tested the full range of cognitive processes within the *Understand* cognitive domain of the Bloom's Revised Taxonomy (Anderson 2001). This proved more difficult than anticipated. While the revised taxonomy was a fertile source of ideas for generating questions, once a question was written, it was sometimes difficult to formally place it within the revised taxonomy. The examples given by the taxonomy's authors are not easy to translate into the programming domain. In many cases the categories within the knowledge domain, did not readily fit with concepts and tasks required in computer programming. It was difficult to match the cognitive tasks undertaken for each question with Bloom's cognitive processes. This resulted in the working group initially categorising most of the questions within the relatively low cognitive level of *Understand*.

3.2 Bloom Categorisation, Version 2

At the group's second meeting, it was realised that the authors had initially categorised the questions according to what we thought the students would do when attempting the questions.

A review of our categorisation was undertaken over three sessions by a consensus between six members of the working group. This recategorisation assumed that the Bloom categories represented a normative model of good practice carried out by students.

The revised Bloom's categorisation is given in Table 1.

It appears that in our initial categorisation we underestimated the level of cognition required to solve some questions. Consequently, not all the 10 questions are within the *Understand* cognitive domain of the Bloom's Revised Taxonomy (Anderson 2001), as had been our original intention.

A small set of interviews with a group of independent academics that had attempted the study instrument were subjected to a "think out loud" interview, like the interviews conducted by the Leeds group (Lister 2004). The interview transcripts lead us to conclude that the best way to refine the categorisation of questions would be by a post-study review of the original categorisations using interview descriptions of the steps taken to solve each question. This "think out loud" approach was not used with the students in the course of this study but will be used in the future.

Q	**Cognitive Process Categories**	**Cognitive Process Subcategories**
1	Apply	Executing
2	Understand	Comparing
3	Understand	Comparing
4	Understand	Comparing
5	Apply	Executing
6	Apply	Executing
7	Analyse	Differentiating
8	Analyse	Differentiating
9	Apply	Executing
10	Understand	Comparing

Table 1: Revised Bloom's categorisation of the question set

3.3 Study Instrument Localisation Issues

Although many of the questions in this study were originally developed in Java, none of the students who actually attempted the 10 questions were taught Java as their first programming language. It was thus necessary to localise the problem set for each institution so that students would be presented with problems in a language with which they were familiar. It was also decided that each institution would apply their own naming conventions and layout standards to the code presented to the students.

Therefore the instrument was also produced in Delphi, C# and C++ dialects. In the course of devising the new questions, the group noted that each language has its own unique idiomatic and syntactical features, which implied subtle differences in the representation of even rather simple MCQ's. Differences in initialization values for indexes, and in relational operators, had the effect of changing the distracters in several instances. However, this localisation did not result in changes to the logic of the questions or the answer sets.

4 Data Collection

The data collected in the initial phase of this study consisted of student answers to nine MCQs and two short answer questions. Analysis of the short answer to question 11, revealed significant complexities and therefore lies outside the scope of this paper.

One hundred and seventeen students participated in this study. These students had either nearly completed or just completed the first semester of their first programming course. In all cases the MCQs counted towards the student's final grade and were taken under examination conditions.

5 Performance Data Analysis on the MCQs

The analysis of the performance data for the MCQs was undertaken using quartiles so that it would be comparable with the previous study by Lister (2004).

Performance data varied from a normal distribution so the use of quartiles is appropriate for MCQ data categorisation (Figure 1).

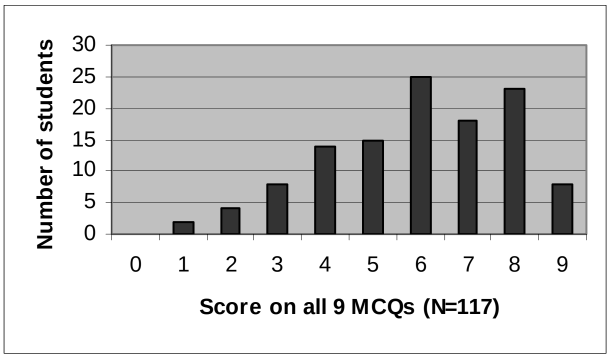


Figure 1: Distribution of scores for students who attempted all MCQs

Analysis of the MCQs was undertaken by establishing quartile boundaries and placing the data into quartiles.

5.1 Performance Data Discussion

Questions 4, 5, 6 and 8 worked well in separating out the more able from the less able students. Most of the distracters worked effectively. So while the difficulty and overall performance varied there was nothing remarkable about these MCQs to report so the analysis of these questions has not been included in this paper.

The two most difficult questions proved to be questions 7 and 8. The Bloom's categorisation of these two questions (Figure 2) identified that they were at the highest level of cognitive processing (analyse) in the MCQ problem set.

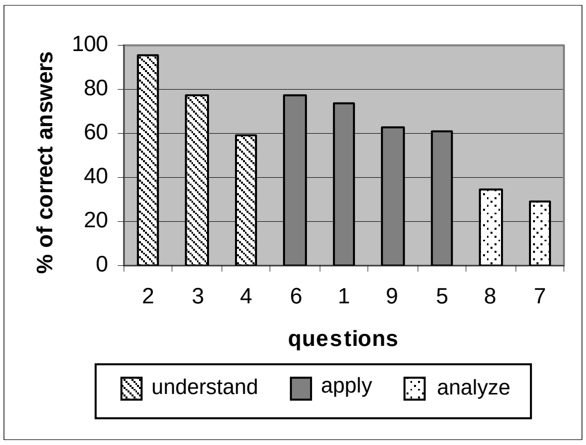


Figure 2: Performance by Bloom's category

5.2 Question 2: The Easiest Question

Over 90% of students answered question 2 correctly. Students were given a piece of code and were required find the matching flow chart. This question was at the lowest cognitive skill level assessed by the set of 10 questions. The full text for question 2 is provided in the appendix of this paper.

MCQs are a common way of testing students in many disciplines, and there is considerable body of literature devoted to the construction and analysis of such tests (Ebel and Frisbie 1986, Linn and Gronlund 1995, Haladyna 1999). A common way of analysing the effectiveness of a MCQ is based upon the notion that MCQs should be answered correctly by most strong students, and incorrectly by most weak students. For question 2, approximately 100% of students in the first quartile (i.e. students who scored 8-9 on all 9 MCQs) answered this question correctly, whereas approximately 80% of students in the bottom quartile (i.e. scored 1-4 on all 9 MCQs) answered this question correctly. On the basis of these two percentages for the top and bottom quartiles, this MCQ is not effective at distinguishing between stronger and weaker students.

A similar but more comprehensive quartile analysis of question 2 is given in Figure 3. This type of figure is an established way of analysing MCQs (Haladyna 1999). It shows the performance of all four student quartiles and also summarizes the actual choices made by students in each quartile. The horizontal axis represents the four student quartiles. The uppermost trend line in that figure represents choice C, the correct choice for Question 2. As stated earlier, approximately 100% of students in the first quartile chose option C. The percentage of students who chose option C was also almost 100% for the second and third as well, but dropped to 80% for the fourth quartile students, where approximately 20% of the students were distracted by option D.

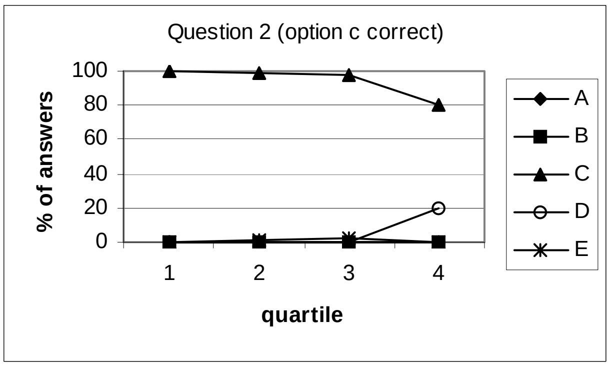


Figure 3: Student responses to Q2, by quartiles[1]

While this question was not effective at distinguishing between strong and weak students, it does at least establish that most students have some minimal grasp of flow control structures in their first programming language.

[1] The quartiles have been numbered from 1-4 where the 1st quartile is actually the top quartile. While this is unusual this identification of quartiles has been adopted in order to allow comparative analysis between this study and the Lister (2004) study.

5.3 Questions 1&9: The Lister Study Questions

Questions 1 and 9 were taken directly from an earlier paper by Lister (2004). In this study similar trend lines (Figure 4, Figure 5) were observed to those recorded by Lister (2004). However, in this study the distracter A for question 9 proved to be a stronger distracter for students in the lower two quartiles.

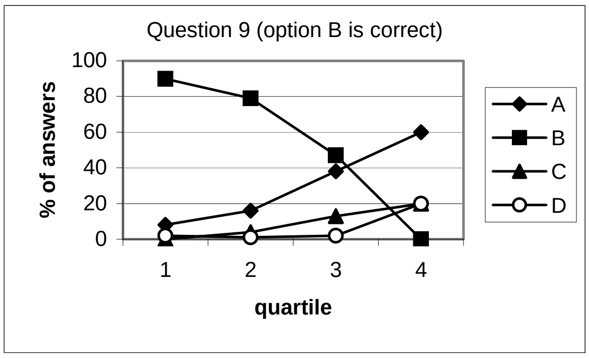


Figure 4: Student responses to Q9, by quartiles

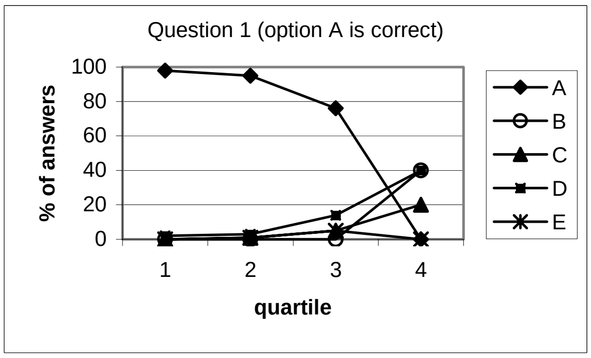


Figure 5: Student responses to Q1, by quartiles

5.4 Question 7: The Hardest Question

The complete text for this question is given in Figure 7. The quartile analysis for this question is given in Figure 6.

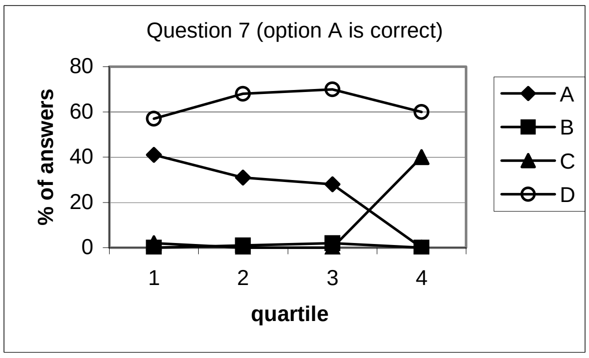


Figure 6: Student responses to Q7, by quartiles

Distracter B was weak and obvious to even the poorest students. Distracter C was only effective on those students who were in the bottom quartile. However distracter D was extremely strong for all students. On the surface, D appears to be correct because by incrementing `iIndex` after the assignment of a value to `iSum` at termination of the loop the desired value of 7 is achieved for `iSum`. Unless students also track the value of `iIndex` they do not discover that the value of `iIndex` after exiting the while loop is not giving the correct value. Perhaps the students assume that it is correct because it is correct in the original question stem code.

Question 7 was intended to make the students operate at a higher cognitive level when it was designed. Figure 2 indicates that it does that and the findings indicate that even the best students had difficulty thinking at this level.

Question 7

The following segment of code was intended to add the elements of the array **iNumbers**, from left to right, until the sum of those elements is greater than the value stored in the variable **iLimit**:

```
int iNumbers[iMAX] = {..some values here..};
int iLimit = 3;
int iIndex = 0;
int iSum = 0;

while((iSum <= iLimit) && (iIndex < iMAX))
{
        iIndex = iIndex + 1;
        iSum = iSum + iNumbers[iIndex];
}
```

The code was intended to finish with the variable **iIndex** containing the first position in the array where the sum exceeds **iLimit**, and **iSum** containing the sum of the array elements from **iNumbers[0]** to **iNumbers[iIndex]** inclusive.

However the given code is buggy. For example, if **iNumbers** has the values { 2, 1, 4, 5, 7}, **iIndex** should be 2 and **iSum** should be 7.

Instead, after the above segment of code is executed, **iIndex** equals 2 and **iSum** equals 5.

The bug in the above code can be fixed by:

a) Replacing iIndex = 0 with iIndex = -1

b) Replacing iSum = 0 with iSum = -1

c) Replacing iSum <= iLimit with iSum < iLimit

d) Moving iIndex = iIndex + 1 from above Sum = iSum + iNumbers[iIndex] to below it

Figure 7: Question 7 of the problem set

5.5 Question 3: The Peculiar Question

Question 2 and question 3 were very similar questions, but the students found question 3 more difficult than question 2. Both questions required the translation of an algorithm or piece of logic from one representation to another. In the case of question 2 (Figure 3) the students were provided with a piece of code and asked to choose the flow diagram that represented the logic of the code.

Two things made question 3 different. Firstly the logic was reversed. Instead of translating from code to a diagram the students were translating from a diagram to code. Secondly the notation of the diagrammatic representation was changed from a flow chart to a structure diagram. One conclusion that may be drawn is that students find structure diagrams harder to interpret than flowcharts. This reflects some of the working group members' experiences when trying to teach algorithmic design to novice programmers using the structure diagram notation.

Quite clearly distracter C was strong for people in the upper middle quartile. The error in C was very minor; an incorrect relational operator was employed in the loop's termination condition. Both A and B contained a bug that we would expect to be harder to identify, the assignment in the body of the loop was incorrect. Additionally, A contained the same error as in distracter C. The bottom quartiles strongest distracter was B. So although they got the termination condition of the while loop correct they overlooked the serious logic flaw caused by the misassignment in the loop body.

Because the stronger students may have perceived this question as 'easy' it can be postulated that the minor error in C was overlooked by many of the students in the upper middle quartile. This bug was not overlooked by those in the lower middle quartile who would, perhaps, have had less confidence and checked all the options before committing to one answer. In the study instrument distracter C was the first option that performed a correct assignment or array copy process. Perhaps the upper middle quartile students selected this option without checking further options to ensure their choice was correct. This means that either distracter C needs to be restructured or simply changed in terms of position on the question sheet.

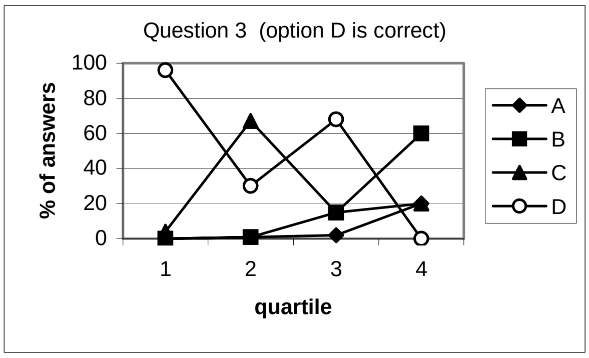


Figure 8: Student responses to Q3, by quartiles

6 Question 10: Summarisation of code

When faced with a code summarisation task, the participants provide a range of responses. Question 10, Figure 9, requires the participants to describe the code in plain English. The responses vary in terms of the precision of the description and the amount of code covered by the description. Some summarisation options may be clearly identified as not correlating with the provided code. The remaining summaries varied in terms of both detail and accuracy with respect to the provided code. A different form of classification was required for this question. The SOLO taxonomy (Biggs 1982, Biggs 1999) provided an approach for performing this analysis.

Question 10

In plain English, explain what the following segment of code does:

```
bool bValid = true;
    for (int i = 0; i < iMAX-1; i++)
    {
       if (iNumbers[i] > iNumbers[i+1])
       {
         bValid = false;
       }
}
```

Figure 9: Question 10 of the problem set

6.1 SOLO analysis categories

To analyse the responses to this question, a series of categories (Table 2) based on the SOLO taxonomy (Biggs and Collis 1982) were developed. As this question provided minimal opportunity to provide an 'extended abstract' response (the highest level in the SOLO taxonomy), this was excluded as an outcome option. The following categories (Table 2) were applied to the responses.

SOLO category	Description
Relational [R]	Provides a summary of what the code does in terms of the code's purpose.
Multistructural [M]	A line by line description is provided of all the code. Summarisation of individual statements may be included
Unistructural [U]	Provides a description for one portion of the code (i.e. describes the if statement)
Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question
Blank	Question not answered

Table 2: SOLO Categories

The student responses above prestructural vary in terms of the amount of code considered in the description (the width) and the extent to which the elements of the code have been related to each other. In the terminology of the SOLO taxonomy, these variations are referred to respectively as the "width" and "depth" of understanding.

The unistructural category includes those responses that focused on only one element of the code. This type of description has a narrow focus with an emphasis on individual statement or part of a statement like "number at index is less than or equal to the number at index plus one". This is a focus on the condition of the “if” statement and ignores the iteration through the array. An example of a unistructural comment, taken from the data collected, is “*If the index number is higher than the index number plus one then the code will be invalid*”.

For the multistructural category, the description describes two or more statements but without showing any relationship between them other than the sequence in which they appear. This category shows an increasing width of analysis. Some variation in depth may be shown through some statements being summarised individually.

Two examples of multistructural comments are:

- "*Compare every element in the array with the one next to it. Return false if it bigger than the one next to it.*"
- "*From the first element in the array to the second last element. Test to see if they are in order from the smallest to the largest If not, return a 'false' to bValid otherwise bValid remains 'true'*"

The extreme form of multistructuralism is when the student describes each line of the code. For example:

- "*bValid is a Boolean. i value is 1. Perform the loop until i equals to the number of array which is deducted -1. If the value of i array is greater than the one of i+1 array, set bValid to "False"*"

The relational category descriptions summarise all of the code. Like the multistructural category, the full width of the code is considered. In this category, the descriptions draw together all of the code showing an increasing depth of understanding of the relationships between individual statements to achieve an overall result. These descriptions of the code segment become more abstract as they rely less on the actual operations specified in the code. A relational description like "checks that the array is in ascending order" is more abstract than a description like “checks that the array elements are ordered from the smallest to the greatest”. These descriptions may include some clarification of what it means to be in “ascending order” such as “two elements being equal are considered as ascending”. Three examples of relational comments are:

- “*Test an array of integers if their values are listed from smallest to largest return true else return false.*"
- “*This piece of code is used to find out if the values in an array are in ascending order* "
- “*To recognise all the elements of the array is in increasing order. If two elements which are neighbours are the same number the program still thinks they are in increasing order*"

The three categories, unistructural, multistructural, and relational could be further subdivided based on the width and depth of abstraction concepts. However, with the quantity of data in this study, it was considered that further dividing of the categories would provide narrow bands that would be difficult to correlate with the responses to other questions.

6.2 Significance of SOLO

It is the contention of the authors that a vital step toward being able to write programs is the capacity to read a piece of code and describe it relationally. A student who can only reason multistructurally may be able to answer a “fixed-code” question correctly, by executing the code, but that student could not write an equivalent piece of code because that student could not translate between the intent of a piece of code and the code itself.

6.3 Analysis of data

The SOLO analysis in this paper is based on a subset of the data collected, the 69 responses collected at one institution. Seven of these were excluded from the SOLO analysis as they were either blank (question was not attempted, three responses) or included a relational type description that was categorised as in error (4 responses). All the blank responses were from students in quartile four. The relational in error responses were distributed over quartiles one, three, and four.

The distribution of responses within the major categories is shown in the following graph (Figure 10). At 54.8%, the multistructural responses are the most common. This may reflect the depth of understanding of the students covered in the study and would be consistent with the view that novices are more focussed on the detail than the patterns represented by the code segment (Chi et al. 1988, Wiedenbeck et al. 1993).

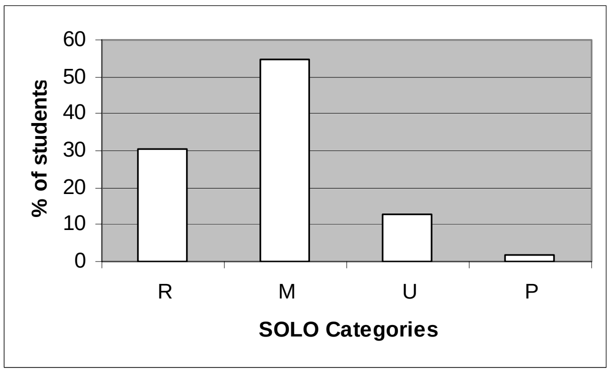


Figure 10: Distribution by SOLO Category

The number of prestructural responses makes it difficult to draw any conclusions related directly with this category unless they are combined with the blank responses.

When compared with the quartile results from questions 1 through 9, the following distribution is obtained (Figure 11). Quartile four favours multistructural responses for Question 10, with a lower unistructural response. All the prestructural and blank responses are also in this quartile.

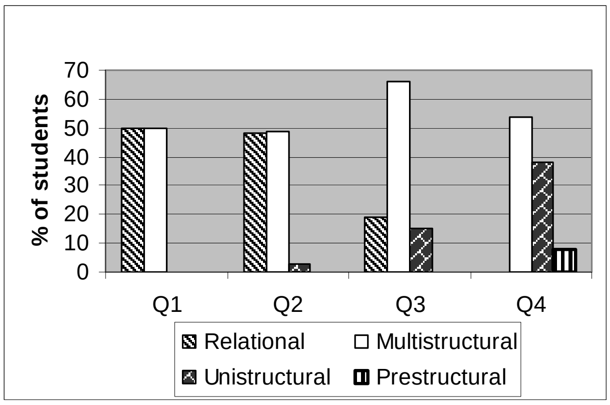


Figure 11: SOLO category distribution by quartiles (Q1 - Q4)

Multistructural responses for Question 10 dominate quartile three, while in quartiles one and two the relational and multistructural responses for Question 10 are even.

Figure 12 depicts the distribution of students within SOLO response by quartile. The relational responses for Question 10 are primarily in quartiles one and two with no relational responses in quartile four. Multistructural responses for Question 10 are fairly evenly distributed but with a slightly higher representation in quartiles two and three. Unistructural responses for Question 10 are primarily in quartile four with none in quartile one.

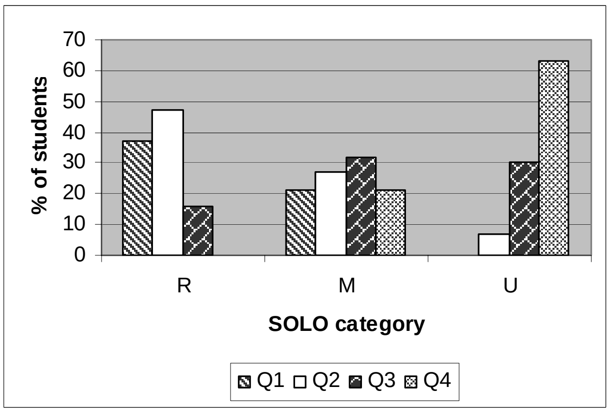


Figure 12: Quartile responses by SOLO category

6.4 Discussion

When applying the SOLO taxonomy (Table 2) in assessment, it is expected that those who demonstrate a relational understanding of the subject have a deeper understanding of the subject matter. The distribution of relational responses shows that they are primarily in the top performing half of the population.

The distribution of the unistructural responses toward quartile four is consistent with the anticipated responses. The slightly higher multistructural responses in the centre two quartiles, was also anticipated although the more even distribution was not. This may be a reflection of the relatively novice status of the respondents with respect to learning to read and write program code.

With the relatively small population in this sample, coding errors may have had an impact on the distributions.

Having only one question that is analysed based on the SOLO categories limits the effectiveness of using these SOLO categories to compare with the MCQ scores. Having more than one question categorised in this manner would have given a better indication of the consistency in the students' abilities to perform in a particular SOLO category.

7 Conclusions

This multi-institutional study into the reading and comprehension skills of novice programmers has reported findings based upon data from three New Zealand institutions. This study has extended prior work by Lister et al (2004), and used a revised set of multiple choice and short answer questions, within two key pedagogical frameworks of analysis. The instrument has tested elements of program comprehension categorised using the revised Bloom framework (Anderson, Krathwohl et al., 2001), and the SOLO taxonomy (Biggs, 1999). The Bloom categorisation has been based upon data from 117 respondents and the SOLO categorisation is based upon a sample of 69 students. While the complete analysis of all data across all participating institutions has not yet been completed, some initial insights and patterns have been observed.

Firstly, categorising programming MCQ's by cognitive complexity applying Bloom's taxonomy, has proven challenging even to an experienced group of programming educators. This may suggest some deficiencies in the Bloom taxonomy when applying it to programming problems, or be a manifestation of the authors current level of understanding of how to apply the taxonomy. It also indicates that assessing programming fairly and consistently is a complex and challenging task, for which programming educators lack clear frameworks and tools.

It appears likely that programming educators may be systemically underestimating the cognitive difficulty in their instruments for assessing programming skills of novice programmers. For non-elite institutions it is likely that some proportion of the high failure rate in introductory programming may be attributed to this difficulty in setting fair and appropriate assessment instruments. In some respects this study echoes the findings of Oliver et al., (2004), whose Bloom's classification of programming courses indicated, for introductory through to intermediate level programming courses, an invariant level of difficulty, which they assessed at the application and analysis level. In contrast they found that the demands imposed by networking courses tended to reside at the lower recall and comprehension levels of the Bloom's taxonomy and did not appear to increase with higher levels of study. Thus the level of difficulty of programming assessments at introductory levels, whether or not inherent in the subject itself, presents a significant and possibly unfair barrier to student success.

The subjects in this study did however perform in a manner consistent with the cognitive difficulty levels, indicated by the assigned Bloom category for each MCQ. This is an encouraging finding as it suggests an ability for educators to apply a "level of difficulty" yardstick with some granularity, to the setting of a programming MCQ.

Analysis of student performance through the SOLO taxonomy did suggest a degree of consistency with the SOLO model, with weaker students less likely to show performance at higher levels of the taxonomy, and stronger students tending to show higher level capabilities. However the results also hinted at the conclusion that novice programmers were not yet able to work at a fully abstract level. The stages through which novice programmers develop to a strongly relational performance level, and the time that this development process may take, needs further investigation and may have been significantly underestimated in many modern computing curricula. Students who cannot read a short piece of code and describe it in relational terms are not well equipped intellectually to write code of their own.

The authors believe that this study provides a more rigorous framework for evaluating performance of novices in programming tasks than the Leeds group study. Through extensions of this study we hope to provide further data to help educators better assess programming comprehension. It is also hoped that by confirming or contradicting the findings emerging from this work, we can deepen our own understandings of how novices learn how to comprehend and write programs.

Acknowledgements

The authorship team would like to thank, in no particular order, Gordon Stegink (formerly of Hope College Michigan, and visiting professor 2003-2004 at Auckland University of Technology), John Hamer (Auckland University), Andrew Luxton-Rielly (Auckland University), Bob Gibbons (Manukau Institute of Technology), Minjie Hu (Tairawhiti Polytechnic), Cathy Saenger (Eastern Institute of Technology), Joy Gasson (Otago Polytechnic) and Hamiora Te Mamo (Waikato Institute of Technology) for their contribution to the initial draft of the study instrument and their helpful discussions. We would also like to thank Gordon Grimsey (Auckland University of Technology) for his assistance in the data collection phase.

8 References

Anderson, L.W., Krathwohl, D.R, Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, R. and Wittrock, M.C.. (Eds) (2001): *A Taxonomy for Learning, Teaching and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*. New York, Addison Wesley Longman, Inc.

Biggs, J. B. (1999): *Teaching for quality learning at University*, Buckingham. Open University Press.

Biggs, J. B. & Collis, K. F. (1982): *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York, Academic Press.

Bloom, B.S., et al.: (1956) T*axonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company.

Bracelet Project http://elena.aut.ac.nz/homepages/staff/J-Whalley/ BRACElet.htm (Accessed October 27, 2005)

Chi, M. T. H., Glaser, R. & Farr, M. J. (Eds.) (1988): *The nature of expertise*, Hillsdale, NJ, Lawrence Erlbaum Associates.

Ebel, R. and Frisbie, D. (1986): *Essentials of Educational Measurement.* Prentice Hall, Englewood Cliffs, NJ.

Haladyna, T. (1999): *Developing and Validating Multiple-Choice Questions (2nd Edition).* Lawrence Erlbaum Associates, Mahwah, NJ.

Linn, R. and Gronlund, N. (1995): *Measurement and Assessment in Teaching*. Prentice Hall, Upper Saddle River, NJ.

Lister, R., Adams E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppällä, O., Simon, B. and Thomas, L. (2004): A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGSCE Bulletin*, **36**(4):119-150.

McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz, (2001): A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bulletin*, **33**(4):125-140.

Oliver, D., Dobele, T., Greber, M., & Roberts, T. (2004): This Course Has a Bloom Rating of 3.9. *Proc. of the sixth conference on Australian computing education*, Dunedin, New Zealand, **57**: 227 - 231.

Wiedenbeck, S., Fix, V. & Scholtz, J. (1993): Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies*, **39**: 793-812.

Appendix

Questions were given to the students with one complete question per page. Here some of the indenting and formatting has been changed in order to fit the journal format.

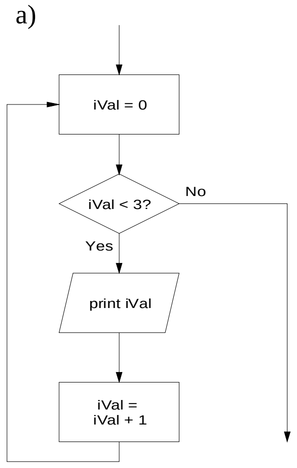
Question 2

Consider the following segment of code:

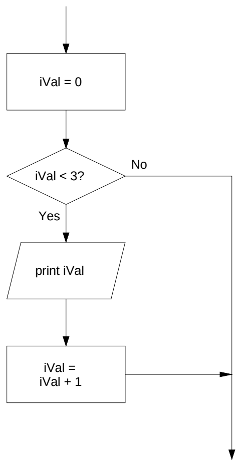
```
const int iMAX = 3;
int iVal = 0;
while (iVal < iMAX)
{
        cout << iVal;
        iVal = iVal + 1;
}
```

Which of these flowcharts represents the logic of this code?

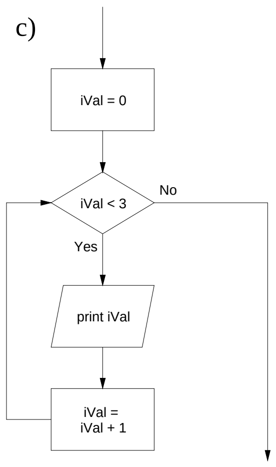
a)



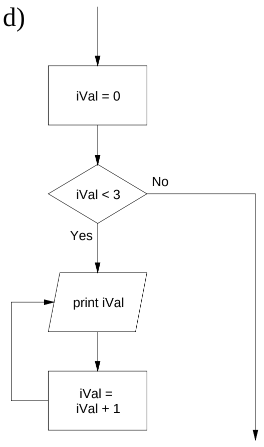
b)



c)

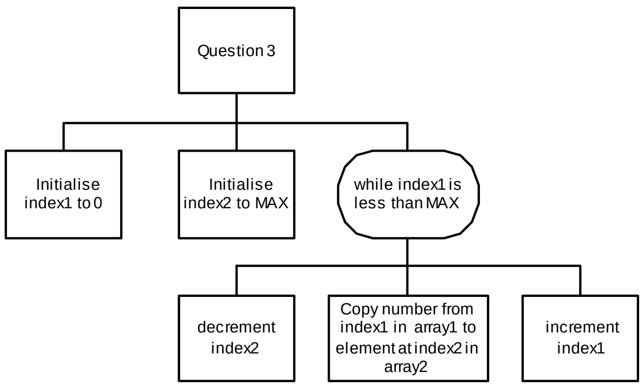


d)



Question 3

Study the structure diagram:


Question 3
Initialise index1 to 0
Initialise index2 to MAX
while index1 is less than MAX
decrement index2
Copy number from index1 in array1 to element at index2 in array2
increment index1

Array1 and Array2 are both arrays containing MAX integers. Which of these code segments correctly implements the logic shown in the above diagram?

a)
```
iIndex1 = 0;
iIndex2 = iMAX;
while (iIndex1 <= iMAX)
{
  iIndex2--;
  iArray1[iIndex1] = iArray2[iIndex2];
  iIndex1++;
}
```

b)
```
iIndex1 = 0;
iIndex2 = iMAX;
while (iIndex1 < iMAX)
{
  iIndex2--;
  iArray1 [iIndex1] = iArray2 [iIndex2];
  iIndex1++;
}
```

c)
```
iIndex1 = 0;
iIndex2 = iMAX;
while (iIndex1 <= iMAX)
{
  iIndex2--;
  iArray2 [iIndex2] = iArray1 [iIndex1];
  iIndex1++;
}
```

d)
```
iIndex1 = 0;
iIndex2 = iMAX;
while (iIndex1 < iMAX)
{
  iIndex2--;
  iArray2 [iIndex2] = iArray1 [iIndex1];
  iIndex1++;
}
```