Salient elements in novice solutions to code writing problems

Jacqueline Whalley†, Tony Clear†, Phil Robbins†, and Errol Thompson*

†School of Computing and Mathematical Sciences
AUT University
PO Box 92006, Auckland 1142, New Zealand
{jwhalley,tclear,probbins}@aut.ac.nz

*School of Computer Science
The University of Birmingham
Birmingham, B15 2TT, United Kingdom
kiwiet@acm.org

Abstract

This paper presents an approach to the evaluation of novice programmers' solutions to code writing problems. The first step was the development a framework comprised of the salient elements, or programming constructs, used in a set of student solutions to three typical code writing assessment problems. This framework was then refined to provide a code quality factor framework that was compared with an analysis using the SOLO taxonomy. We found that combining our framework with the SOLO taxonomy helped to define the SOLO categories and provided an improved approach to applying the principles of SOLO to code writing problems.

Keywords: SOLO taxonomy, novice programmers, assessment.

1 Introduction

This paper furthers one aspect of the work of the ITiCSE 2009 Paris working group (Lister et al., 2009) in classifying novice student responses to program code writing exercises. The aim of classifying student responses is: 1) to better understand typical patterns of response; 2) to better understand how students typically approach code writing questions; 3) to derive repeatable measures of novice student performance on assessment tasks; 4) to apply a pedagogically accepted theoretical framework to this investigation; 5) to identify areas where students commonly experience difficulty.

As a result of the above insights we would hope eventually to: 1) develop more effective teaching and learning strategies; 2) use this knowledge in order to develop more consistent expectations of novice student performance; and 3) to assist in the design of fair and appropriate assessment instruments in examinations and tests.

2 Background

The Paris working group (Lister et al., 2009) applied recognised educational frameworks (Bloom and SOLO) as classification schemes for mapping examination questions and novice student responses. The work of the group extended from code comprehension questions to initial attempts to address code writing questions. These attempts adopted a top down strategy in order to directly map each student response to a SOLO level. While they achieved a mapping for three very distinct questions and came to a consensus between four raters, the process seemed very context bound and question specific. In subsequent work we have revisited this approach and some of the underlying assumptions.

In this paper by contrast, we begin with the student response as raw data and build from that basis in a grounded manner to empirically derive a framework from the students' own work, before attempting as a subsequent step a SOLO classification of the responses given.

3 Methodology

Analysis of program understanding has been defined as "identifying artefacts and understanding their relationships; this process is essentially pattern matching at different abstraction levels" (Tilley 2000). More recently Meerbaum-Salant et al., (2010) have used a form of content analysis (Stemler 2001) of written text in a study which systematically analysed students' written code, and categorised their solutions according to the SOLO and Bloom taxonomies.

The study we report here, while adopting similar methods to *content analysis*, is better defined as a study applying *grounded theory*. Grounded theory (GT) is a method for empirically deriving theory from data, typically through applying an inductive and rigorous process of coding and categorisation. GT was originally conceived as an analytical and conceptual, creative process of constant comparative coding by Glaser and Strauss (1967). They have asserted that "grounded theory allows no speculation…one can be just as systematic with qualitative data as with quantitative data" (p. 200) and in generating theory the key position is that "the theory should fit the data" (p. 201) and not vice versa.

Given our goal of starting with the students' own code to derive the underlying patterns, GT presented an appropriate research method. A bottom up approach to the analysis of student responses to three code writing questions was undertaken. In outlining strategies for GT analysis Glaser and Strauss (1967, pp. 62-63) made the following distinctions between "sampling strategies":

> *"It is important to contrast theoretical sampling based on the saturation of categories, with statistical (random sampling). Theoretical sampling is done in order to discover categories and their properties, and to suggest the interrelationships into a theory."*
>
> *"The adequate theoretical sample is judged on the basis of how widely and diversely the analyst chose his groups for saturating categories according to the type of theory he wished to develop."*

Copyright © 2011, Australian Computer Society, Inc. This paper appeared at the 13th Australasian Computer Education Conference (ACE 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 114. J. Hamer and M. de Raadt, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

To simplify this initial investigation we examined only solutions that were complete and would compile and run. All other solutions were incomplete and showed that students lacked knowledge of basic programming constructs. Therefore SOLO analysis of seriously flawed responses would give inconsistent categorisations. The unit of analysis was the segment of code comprising the student response to a selected question. A subset of solutions was selected for analysis, comprising questions which exercised different programming constructs. We believe that we achieved saturation in the programming constructs from the subset we chose (see Section 4). Student responses were selected from questions that were posed to elicit responses at different SOLO levels. The goal was to establish a set of empirical categories comprised of salient programming elements. The student responses were coded to reflect the primarily syntactic categories of salient programming elements that emerged from the constant comparative analysis of the data.

These salient elements coded at the syntactic level were subsequently condensed through a form of feature extraction into broader concepts or 'code quality' categories. Those categories were used to represent the patterns of code used by novice programmers.

The students' responses were subsequently classified independently, by three researchers, using the SOLO taxonomy. We were interested to see if an existing taxonomy would be sufficient or if the salient elements and quality factors made it easier to measure the level at which a student was answering code writing questions.

The three researchers then met to reconcile the differences in the resultant SOLO codings and to achieve a consensus on their ratings, based upon agreeing common interpretations of the definitions in Table 1. This session was further informed by codings from an additional researcher working remotely. Once a consistent set of understandings had been derived a basis for presenting the findings of this study was achieved.

3.1 The SOLO taxonomy

The SOLO taxonomy describes levels of increasingly integrated thinking in a student's understanding of a subject, through five stages. Biggs (1999) describes the types of verbs that apply for each of the levels of the taxonomy (p. 47) and provides an example of ordering outcome items by the taxonomy (pp.176-178). These levels are placed into two phases suggesting that learning passes through various stages from a more quantitative phase (surface) to a more qualitative one (deep, connecting and relating ideas) as learning tasks and their complexity increase. Hattie and Purdie (1998, p. 156) provide a number of examples of the use of the SOLO taxonomy.

More recently SOLO has been used to reliably classify code reading questions and the student responses to those questions (Clear et al., 2008; Sheard et al., 2008). An initial set of guidelines and descriptors for using SOLO to classify student code writing solutions (Table 1) were proposed by Lister et al. (2009). These descriptions are the ones that we initially employed to independently classify the student responses to the questions discussed here.

When the SOLO taxonomy is applied to short segments of code, the learner needs to have more than a working solution; they need to show an understanding of the types of constructs that best implement the solution, to utilise program structures that communicate the intent of their code to others, and produce code that is easy to maintain.

Additionally the phrasing of the problem itself is critical to the possible SOLO level for a response. In analysing code writing tasks, it was found that the nature of the question had an impact on the type of solutions that were possible (Lister et al. 2009). Some questions did not allow for much more than a direct translation into the programming language while others allowed for greater interpretation. This supports the view that a question can be posed to elicit responses at a given SOLO level. The categories defined by Lister et al. (2009) were based on an example for language translation from Hattie and Purdie (1998). Hattie and Purdie (1998) argue that the shift through the SOLO levels shows an increasing understanding of how the phrase should be interpreted rather than just translated. This shift shows an increasing awareness of the relationship between the words and how that relationship communicates meaning.

Phase	SOLO category	Description
Qualitative	Extended Abstract – Extending [EA]	Used constructs and concepts beyond those required in the exercise to provide an improved solution
	Relational - Encompassing [R]	Provides a valid well structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Quantitative	Multistructural - Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.
	Unistructural – Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
	Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 1: SOLO categories for code writing solutions

Although Table 1 represents the SOLO categories as though there were distinct boundaries, it should be recognised that what is represented is actually a continuum (Thompson 2010, Meerbaum-Salant et al. 2010). Supporting this view, a statistical analysis of SOLO levels for code comprehension questions conducted by the Paris working group (Lister et al., 2009), confirmed the ordinality of the SOLO scale.

4 Analysis of student code writing solutions

The data consisted of exam questions and students' answers in programming courses offered in New Zealand and Finland. Three exams were used from introductory

programming courses. The answers of nearly 750 students were available for analysis. The programming languages covered were Java, Perl and Python. All of the exams included code tracing questions, most included code-explaining, code writing questions and Parsons questions. As noted in Section 3 above a theoretical sampling strategy was adopted with a proportion of the answers from each examination chosen for further analysis. Three typical CS1 code writing problems were selected from the above corpus as representative of different programming constructs and a progression of technical difficulty. What follows is a detailed discussion of the analysis of each of these three questions applying the methodology described in section 3.

4.1 Discount

The discount problem was taken from a written examination for first semester (CS1) students where the languages used were Python and Finnish. A subsample of valid student responses was analysed. Forty eight responses were analysed and exemplars of typical student responses are given in Table 2 (refer to the appendix for the less common code pattern examples, we have chosen to leave these in Finnish as these constitute the raw data).

A shop gives reductions of the prices as follows: if the original price of an item is at least 100 Euros but less than 200 Euros, the reduction is 5%. If the original price is at least 200 Euros then the reduction is 10%.

Pattern	Typical Code Example

3	def main();
	  hinta = raw_input("Anna alkuperainen hinta.");
	  hinta = float(hinta);
	  if hinta >= 200:
	  hinta = 0.90*hinta
	  elif hinta >= 100:
	  hinta = 0.95*hinta
	  print "Hinta Alennettuna:", hinta
	main()

5	def main();
	  rivi = raw_input("Anna alkuperainen hinta.");
	  hinta = float(rivi);
	  if hinta >= 200:
	  uusihinta = 0.90*hinta
	  if hinta >= 100 and hinta <200:
	  uusihinta = 0.95*hinta
	  if hinta < 100:
	  uusihinta = hinta
	  print "Tuotteen alennettu hinta on:", uusihinta
	main()

6	def main();
	  rivi = raw_input("Anna alkuperainen hinta.");
	  alku_hinta = float(rivi);
	  if alku_hinta < 100:
	  print "Hinta on", alku_hinta, "euroa"
	  elif alku_hinta <200:
	  hinta = 0.95*alku_hinta
	  print "Alennettu hinta on", hinta, "euroa"
	  elif alku_hinta <200:
	  hinta1 = 0.90*alku_hinta
	  print "Alennettu hinta on", hinta1, "euroa"
	main()

Table 2: Example of the most prevalent 'discount' solution patterns

Applying a grounded theoretic strategy a set of empirical codes were derived based primarily upon the syntactic constructs present within the student responses. This resulted in the set of salient elements portrayed in Table 3. The syntactic elements are shown in the second column of the table with one broader grouping based on function or purpose of the code in column one. An abstraction beyond the salient elements is given in column three where key features have been extracted based on the contribution of the salient element to the quality of the end code.

As can be extrapolated from Table 3, focusing on the selection function, solutions that used two selection clauses were preferable and those with three selection clauses had one unnecessary clause. The selection function class was therefore split into two quality factors without or with redundancy respectively. With reference to the printing and calculation functions better solutions had one print or calculation statement outside of the selection statement to remove code repetition. If a discount subroutine was written then the solution to the discount calculation was also judged to be a generalised solution.

Function	Element	Quality Factor
selection	if/else if	no redundancy
	2 x if	
	if/else if/ else	redundancy
	if/else if /else if	
	3 x if	
	& or | used	
discount calculation	in a subroutine	generalised
	in each selection clause	redundancy
	one calculation	no redundancy
printing	in each selection clause	redundancy
	one statement **after** selection functionality	no redundancy

Table 3: Salient element framework for the 'discount' problem

In the next step, Table 3 has been condensed to allow us to map student code to the function and quality framework. Using this framework, six patterns of code construction were observed.

Function	Quality Factor	Code Pattern					
		1	2	3	4	5	6
selection	no redundancy	X	X	X			
calculation	generalised	X					
	no redundancy		X		X		
printing	no redundancy	X	X	X	X	X	
Number of solutions		1	1	6	5	24	11
SOLO Classification		M	U	U	U	U	U

Table 4: Refined quality framework for the 'discount' problem

These code patterns are mapped against their respective functions and quality factors in Table 4. The

concepts extracted in this model then enabled us to conduct a mapping of each response pattern to the SOLO taxonomy.

Despite the large variation in responses, for a relatively simple question, the differences were in the minor detail. Perhaps this is a function of the limited level of complexity of the question which essentially just assesses ability to frame conditional statements.

This question in terms of the SOLO taxonomy is posed at the *multistructural* level. It requires some interpretation to arrive at a suitable solution (cf. Table 1) but a significant part of the specification may be directly translatable into a solution.

The sequence of selection statements in the majority of cases were found to be 'a direct translation of the specification' and therefore coded as *unistructural*. However in some cases the sequence had been reordered away from a direct translation of the specification but this reordering provided a less integrated solution than would have been provided by a direct translation response (pattern 3). In other cases a solution had been improved in one 'quality factor' aspect, e.g.: removed repetition of the printing statement, but had introduced redundancy by double checking a boundary value.

One student wrote a generalised subroutine to calculate the discount and then used that routine in the main method (pattern 1, Table 4). This response was the only response observed that was not coded as *unistructural* because 'the code had been reordered to make a valid solution' (cf. Table 1). Since the code in the subroutine was close to 'a direct translation of the specification', although a more sophisticated response, we chose not to code this at the *relational* level. Using the quality factors as a guide we were able to place the student responses along the SOLO continuum (Figure 1).

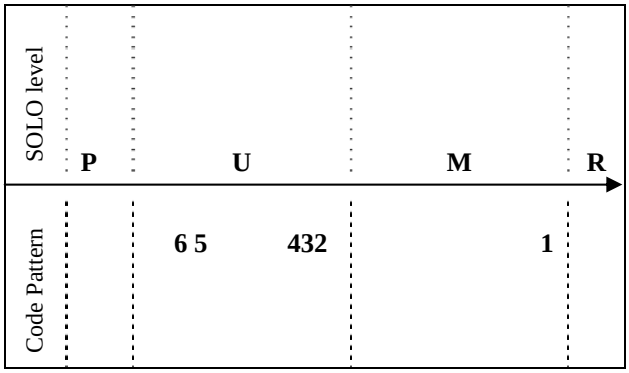

Figure 1: Mapping to SOLO for 'discount' using quality factors

Supporting the notion of a continuum the code patterns 4, 3 and 2 lean a little more towards a *multistructural* level response. Code pattern 1 by contrast leans towards a *relational* response because of the use of a subroutine which is a more integrated solution but in this case is still 'close to a direct translation' of the specification (cf. Table 1).

4.2 Average

This question was taken from a second semester (CS1.5) programming class, with a focus on scripting languages. The question, given below, was part of a practical exam.

Write a Perl script to allow students to calculate their average exercise mark for a semester. The script should:

1. *Prompt the student for their name.*
2. *Prompt for the next exercise mark and allow it to be input*
3. *Do this for 5 marks.*
4. *Display the student's name, and their average mark.*

Extra credit will be given if your script contains a sensible subroutine that is correctly used.

Typical code patterns are depicted in Table 5 with a focus on the subroutine function or its absence (Table 6) where the key variations were apparent.

Pattern	Typical Code Example

1	sub findAve (){
	$_[0] / $_[1];
	}
	for (my $i = 0; $i < 5; $i ++){
	print "Enter next mark: \n";
	chomp (my $mark = <STDIN>);
	$total+= $mark;
	}
	my $average = &findAve ($total,5);

2,4,5	sub findAve (){ $_[0] / 5; }

6	my $i = 0;
	my $total = 0;
	for ($i=0, $i<5, $i++){
	print "Enter next mark: ";
	chomp($total += <STDIN>);
	}
	my $average = $total/5;
	print "Average : $average\n";
	exit;

7	print "Please enter first mark \n";
	my $mark1 = <STDIN>;
	print "Please enter second mark \n";
	my $mark2 = <STDIN>;
	print "Please enter third mark \n";
	my $mark3 = <STDIN>;
	print "Please enter fourth mark \n";
	my $mark4 = <STDIN>;
	print "Please enter fifth mark \n";
	my $mark5 = <STDIN>;
	my $total = 0;
	my $total1 = $total + $mark1;
	my $total2 = $total1 + $mark2;
	my $total3 = $total2 + $mark3;
	my $total4 = $total3 + $mark4;
	my $total5 = $total4 + $mark5;
	my $average = $total5/5;
	print "Total: $total5 \n";
	print "Average: $average \n";

Table 5: Example of the most prevalent 'average' solution patterns

The initial salient element analysis for the averaging problem is provided in Table 6. Solutions that did not use a loop to get the user input had code repetition and were considered to have high redundancy. One student wrote a cohesive, generalised subroutine to get the user input. But

the majority of students wrote a subroutine to calculate the average.

The elements identified with their respective functions and quality factors are depicted in Table 6. We can see in this question while redundancy and generalisation are quality factors in common with the 'discount' problem, the requirement for a subroutine has added a new design dimension of cohesion of the subroutine design.

Function	Element	Quality Factor
input	Uses a loop	low redundancy
	No loop	high redundancy
	In a subroutine	generalised
Has subroutine	Sub uses a parameter	generalised
	Also gets input	low cohesion
No subroutine	Sums in a loop	low redundancy
	Sums without loop	high redundancy

Table 6: Salient element framework for the 'average' problem

Table 7 extends the salient element framework to highlight the varying code patterns relating to the identified function and quality factors.

Function	Quality Factor	Code Pattern						
		1	2	3	4	5	6	7
sub	used	X	X	X	X	X		
	generalised	X		X				
	high cohesion	X			X	X		
number	generalised		X		X			
input	no redundancy	X	X	X	X	X	X	
Number of solutions		20	1	1	1	2	14	1
SOLO classification		M	M	M	M	M	U	U

Table 7: Refined quality framework for the 'average' problem

As Table 7 shows there was some variation in the way that students wrote their subroutines but with two dominant patterns suggesting a bimodal response pattern based upon a student's decision to write a subroutine. The primary pattern, pattern 1 (cf. Table 5 for sample code) is an example of a response that used a generalised subroutine. In contrast patterns 2, 4 and 5 did not use a generalised subroutine (also cf. Table 5). Pattern 1 took two parameters, the total and the item count, whereas pattern 3 took one, an array containing the input data. The latter resulted in a solution without cohesion as it consisted of one large subroutine that read input and calculated the average. Only two students used a variable to hold the number of items (patterns 2 and 4) instead of hard coding the '5' into the code. The one example where the solution has redundancy in getting the input did not use a loop. More common was the use of a loop for input but no subroutine at all (pattern 6).

While the patterns of response were divided bi-modally at the *unistructural* and *multistructural* SOLO levels the question itself is clearly set at a multistructural level (cf. Tables 1 and 7). Parts of the problem were directly translatable in that students were given the steps of the algorithm required for its solution. The question offered an extra mark for a simple subroutine, without specifying what it had to do, so some interpretation was required for this.

One student wrote a solution that took the line "Do this for 5 marks" to mean repeat the input line 5 times. Most realised that the line was meant to indicate that a loop was required, especially as the accompanying marking scheme clearly stated that a loop was needed.

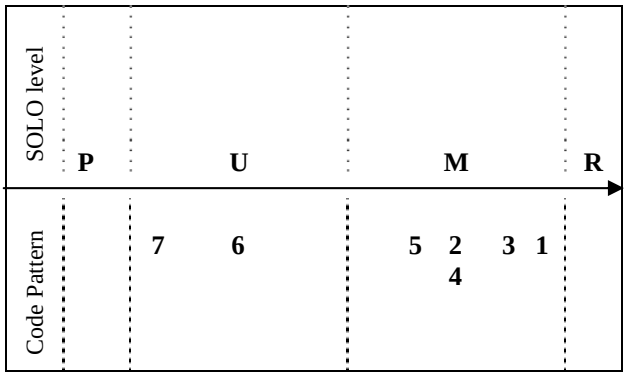

Figure 2: Mapping to SOLO for 'average' using quality factors

We classed all solutions without a subroutine as *unistructural* since they were no more than a direct translation of the problem description that opted out of writing a subroutine. Next in the continuum (Figure 2) were those answers where a subroutine was used. We considered the inclusion of a subroutine to indicate a *multistructural* response. We did not consider it to be a *relational* response because it was clearly stated in the question that a subroutine was expected. However reflecting the continuum model we can see from Figure 2 that pattern 7 is close to a *prestructural* response and pattern one approaches the *relational* level of SOLO.

4.3 Print a box of asters

This question was taken from the final written exam for a first semester (CS1) java programming class.

The students were asked to write code to print a box of asterisks (*) with the same number of rows and columns, an example was given of a 5 by 5 box of asters. The students were provided with the method signature which had a single parameter that represented the width and height of the box. The majority of the students did not attempt to answer this question. This appeared to be perceived as a difficult question by the students, although students had prior exposure to writing a method that printed a triangle in their lab class.

Table 8 portrays the typical solution code for both the incorrect and correct solutions. In this question many students wrote functioning code that iterated one too many times (code patterns annotated in Table 8 "<=" with one or more loops) or created a box of fixed width or area.

We can see in this question while redundancy and generalisation are quality factors in common with the 'discount' and 'averaging' problems, the increased complexity of the problem has introduced further design issues relating to the degree of connectedness of the code. It has also enabled partially correct working solutions to be provided.

Code Patterns	Typical Code Example

1, 7
2, 8 (<=)
	public void printBox(int size){
	for(int i = 0; i < size; i++){
	for(int j = 0; j < size; j++){
	System.out.print("*");
	}
	System.out.println( );
	}
	}

1, 9
2, 10 (<= 2nd loop)
	public void printBox(int size){
	String sStar = "";
	for(int i = 0; i < stars; i++){
	sStar += "*";
	}
	for(int j = 0; j < stars; j++){
	System.out.println(sStar);
	}
	}

3
4, 11(<= loop)
	public void printSquare(int size){
	for(int j = 0; j < stars; j++){
	System.out.println("*****");
	}
	}

6, 12	public void printSquare(int size){
	for(int j = 0; j < stars; j++){
	System.out.println("*");
	}
	}

5, 13	public void printSquare(int size){
	System.out.println("*****");
	System.out.println("*****");
	System.out.println("*****");
	System.out.println("*****");
	System.out.println("*****");
	}

Table 8: Example of the most prevalent 'box of asters' solution patterns

In refining from this salient element framework (Table 9) to a refined set of code patterns new choices presented themselves. Depending on the learning goals at this level teachers may choose to emphasise different quality factors in assessing student code. Key distinctions in this instance were between generalisability, connectedness (or level of integration) and potential efficiency.

Function	Element	Quality Factor
iteration	nested	generalised & connected
	2 independent loops	generalised & not connected
	1 loop	not generalised
	No loop	not generalised & redundancy
	Uses parameter	generalised
	Terminates correctly	correct solution
	Loop 1 too many	incorrect solution
	Loop 1 too few	incorrect solution
asters	As a local variable	generalised

Table 9: Salient element framework for the 'box of asters' problem

In the course context for this question more importance was placed on generalisability and indeed the question itself required a generalised solution. While in our context this focus on generalisability has been adopted as a design principle in other contexts with a stronger focus on algorithms the efficiency of the solutions maybe a focus. A discussion follows of two refinements of the salient element framework based on two different perspectives of code quality.

The first refinement involved condensing the iteration elements based on the generalisability of the code as a quality factor.

Function	Quality Factor	Code Patterns					
		1	2	3	4	5	6
iteration	generalised	X	X				
	terminates correctly	X		X			X
result	*n* by *n* box, correct size	X					
	an *n* by *n* box, size incorrect		X				
	a 5 by *n* box			X	X		
	A 5 by 5 box					X	
	a line of 5 asters						X
Number of solutions		15	3	4	4	1	4
SOLO classification		R	R	U	U	U	P

Table 10: Refined quality framework based on generalisability for the 'box of asters' problem

Six unique patterns of code were identified when the code was classified using the quality framework based on generalisability of the print a box of asters method.

If the solutions were able to produce a box of equal width and height of any size (the answer utilised the parameter supplied) then the iteration was considered to be a more generalised solution (see code patterns 1 and 2, Table 10 and code examples Table 8). Those students who chose to answer this question tended to provide a correct generalised solution. The few solutions that were not generalised usually either printed a fixed width box or a single vertical or horizontal line of asters and also gave incorrect output.

Function	Quality Factor	Code Patterns						
		7	8	9	10	11	12	13
iteration	connected	X	X					
	terminates correctly	X		X			X	
result	*n* by *n* box, correct size	X		X				
	an *n* by *n* box, size incorrect		X		X			
	a 5 by *n* box					X		
	A 5 by 5 box							X
	a line of 5 asters						X	
Number of solutions		10	3	8	4	5	6	1
SOLO classification		R	R	R	R	U	P	U

Table 11: Refined quality framework based on connectedness for the 'box of asters' problem

The second refinement attempt, in Table 11, condensed the categories on the basis of the degree of connectedness (or degree of integration) of the code. In

solutions that use two sequential loops the response had two somewhat disconnected pieces of code. While it could be argued that this solution has some connection between the two loops because the second loop uses the string built by the first loop (cf. Table 8), it is possible to give a more connected answer by nesting the loops.

This problem is considered to be posed at a *relational* level. It provides the learner with a description of the problem to be solved. The description is complete in the sense that it describes fully the requirement from the perspective of the problem domain but provides only minor clues as to how the problem might be programmed.

Since the algorithm was not provided there is no method of clearly defining what a direct translation solution (*unistructural*) would look like. For this reason it is not possible for students to provide a correct solution that can be considered to be *unistructural* or *multistructural*.

Across all 13 patterns identified in Tables 10 and 11 for the quality factor of correctness, there were three code patterns observed that provided a correct solution (cf. Table 11, patterns 7 and 9 and Table 10, pattern 1). The other answers failed to output a generalised solution and did not provide a correct solution (typically by incorrectly terminating the loop). For the purpose of this SOLO analysis we ignored the loop termination bug as it is not significant when considering the SOLO class of the answer (which assesses the level of integration of a response).

The first correct method (Table 8, patterns 1 and 7) used a nested loop which provided a connected and generalised solution. The nested loop solution is less efficient (O(n^2)) than the second generalised solution that uses somewhat disconnected sequential loops (O(n)) (Table 8, patterns 1 and 9). These solutions are considered to be *relational*.

Code patterns 5 and 13, 3, 4 and 11 provide a direct translation of the exemplar given in the question and can only ever produce a 5 by 5 or a 5 by *n* box of asters. Some of these code patterns use a loop rather than five sequential print statements and are therefore considered to be slightly better solutions. This subtle difference is illustrated in Figure 3 along a SOLO continuum. However all student solutions following these patterns fail to provide the required generalised solution and were classified as a *unistructural* response.

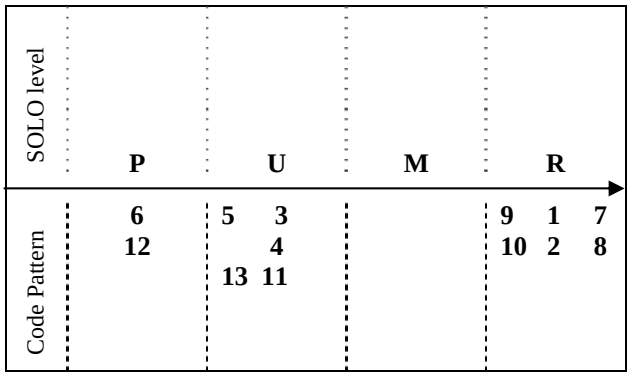

Figure 3: Mapping to SOLO for 'box of asters' using quality factors

Solutions that printed a row or column of asters, patterns 6 and 12, show that the students recognise that a loop is required to solve the problem but do not really grasp how that loop functions and their solutions were therefore coded as *prestructural*.

5 Conclusion

In previous research, mapping from student code to the SOLO taxonomy has proven difficult (Clear et al., 2008, Lister et al., 2009), since the mapping process seems very context bound and question specific. Therefore achieving consistent ratings is challenging, especially for code writing problems. In this study a grounded approach has been adopted to work from student responses bottom up through a two layer coding and concept mapping process. Based upon the resulting refined quality framework we have been able to identify critical elements, which can contribute to more consistent and supportable SOLO categorisations of novice programming students' responses to writing questions. A depiction of the empirically grounded mapping process is given in Figure 4.

We believe that this salient element and quality framework helps to define the SOLO categories and provides a novel way of matching the principles of SOLO for code writing problems. The process of identifying salient elements at the syntactic level should be readily reproducible for chosen problems by knowledgeable CSED researchers. In the feature extraction stage there are some basic features that are replicable and discernable across different code writing questions given to CS1 students. These features encompass the degree of redundancy, efficiency, generalisability and integration observed in the solution code. In some cases degree of coupling and cohesion also play a role. The features represent abstractions from the code itself, based upon qualitative judgments, which can be adapted depending upon the design goal for the code writing exercise.

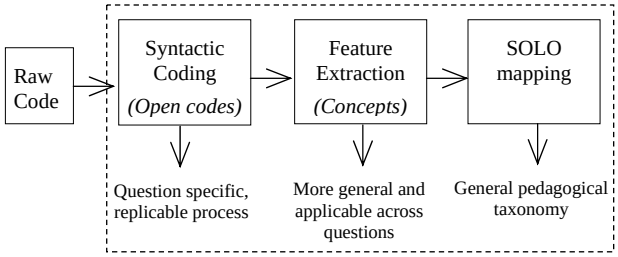

Figure 4: The mapping process

Further study needs to be undertaken that allows us to establish the reliability of the suggested coding practice. Potentially multi-rater studies could be conducted (Clear et al., 2008).

The study has caused us to suggest refinements to prior work. In coding the averaging problem (Section 4.2) we observed students responses that highlight an issue with the description provided by Lister et al. (2009) for *multistructural* responses to code writing questions. Therefore a revised version of Table 1 is provided below to include this refined *multistructural* response definition.

Phase	SOLO category	Description
Qualitative	Extended Abstract – Extending [EA]	Used constructs and concepts beyond those required in the exercise to provide an improved solution
	Relational - Encompassing [R]	Provides a valid well structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Quantitative	Multistructural - Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make *a more integrated and/or valid solution.*
	Unistructural – Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
	Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 12: Refined SOLO categories for code writing solutions

The previous definition stated that a *multistructural* response represented a translation that is close to a direct translation. The code may have been reordered to make a 'valid' solution. Yet in the averaging problem we saw responses that were a slightly reordered translation and that were correct valid solutions, but the reordering led to a less integrated solution. Such translations, while often still at the *multistructural* response level, tended towards the *unistructural*.

In our experience, educators need to be careful when applying SOLO to classify student responses. It is easy to lose sight of the intention of SOLO. *Unistructural* means focusing on a single concept or salient element. *Multistructural* focuses on multiple concepts or salient features but not integrating them all. *Relational* requires seeing all the salient elements or features and utilising them (i.e. seeing the relationships between the concepts and features). In this study these terms have been interpreted through Table 12 above, and the process of assigning a SOLO level to a student response has involved an assessment of the degree of distance between the answer and the specification. Thus the judgment for the code writing process relates to the level of translation of the specification demanded to implement the code. This reflects the level of abstraction or integration of thought required by the student.

The insights from this study may further serve to explain the contradictory findings of Meerbaum-Salant et al., (2010), who noted when applying a combined BLOOM and SOLO taxonomy, that students performed less well on a lower level *Multistructural Creating* task than on a deemed higher level *Relational Applying* task. Lopez et al., (2008) would also support that view in suggesting that code writing is a higher order skill than tracing – the *Relational Applying* task in the Meerbaum-Salant et al., (2010) study. But perhaps the code writing/reading distinction is more subtle than a simple hierarchy. In the latter study we surmise that the authors have not taken into account the degree of translation demanded by the code writing task, but posited a SOLO level for the question 'in the abstract' based upon it requiring the combination of an assumed sequence of steps. Such tasks may indeed involve more integration of thought than implied by a *multistructural* classification, which requires some degree of re-ordering and integration of thought (cf. Table 12), perhaps placing it closer to the *Relationa*l level on the SOLO continuum.

To conclude we believe the salient element framework presented in this paper should serve as an aid to CSED Researchers and CS Educators seeking to analyse novice programmer responses to code writing questions, and to map them consistently to a SOLO level. By doing so we intend that our understanding of the programming process exercised by novice programmers may be deepened, and that we may build a greater awareness of what reasonable expectations may be set for novice performance on code writing tasks. The intended impacts from this deeper, research derived, understanding are: more consistent and equitable designs for code writing questions, an improved learning experience for the novice and an overall increase in the quality of teaching and assessment of novice programmers.

6 Acknowledgements

We wish to express our thanks to members of the ITiCSE 2009 Paris working group and especially Otto Seppälä who was a member of the subgroup who provided data for one of the questions and helped with assigning SOLO categories to the responses.

7 References

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

Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., et al. (2008): Reliably Classifying Novice Programmer Exam Results using the SOLO Taxonomy. S. Mann and M. Lopez (eds.), *Proc. of the 21st Annual NACCQ Conference*, **1**: 23-30. Auckland, New Zealand: NACCQ.

Glaser, B. and Strauss, A. (1967): *The Discovery of Grounded Theory*. Mill Valley, CA: Sociology Press.

Hattie, J. and Purdie, N. (1998): 'The SOLO model: Addressing fundamental measurement issues', in Dart, B. and Boulton-Lewis, G. (eds.), *Teaching and Learning in Higher Education.* 145–176. ACER Press.

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., et al. (2009): Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bull.* **41**(4): 156-173.

Lopez, M., Whalley, J., Robbins, P. et al., (2008): Relationships between reading, tracing and writing skills in introductory programming. M. Caspersen, R. Lister and M. Clancy (eds.), *Proc. of the Fourth International Computing Education Research*

Workshop (ICER 2008). 101-112. Sydney, Australia: ACM.

Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. (2010): Learning Computer Science Concepts with Scratch. K. Sanders, M. Caspersen and M. Clancy (eds.), *Proc. of the Sixth International Computing Education Research Workshop (ICER 2010)*. 69-76. Aarhus, Denmark: ACM.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E. and Whalley, J. L. (2008): Going SOLO to assess novice programmers. *SIGCSE Bull.*, **40** (3): 209-213.

Stemler, S. (2001): An Overview of Content Analysis *Practical Assessment, Research & Evaluation*, **7**(17). Retrieved August 16, 2010 from http://PAREonline.net/getvn.asp?v=7&n=17

Thompson, E. (2010): Using the Principles of Variation to Create Code Writing Problem Sets. To appear in *Proc of the 11th Annual Conference of the Subject Centre for Information and Computer Sciences*, Durham, UK, http://www.ics.heacademy.ac.uk/

Tilley, S. (2000): The canonical activities of reverse engineering. *Annals of Software Engineering*, **9**: 249-271.

Appendix

This appendix provides example code for the less frequent code patterns observed for the discount pattern.

Pattern	Typical Code Example

1	def main();
	print "Losken tuotteen alennettu hinnan."
	rivi = raw_input("Anna tuotteen hinta.\n")
	hinta = float(rivi)
	aleenushinta = laske_alennelta_hinta(hinta);
	print "Tuotteen alennettu hinta on:", uusihinta, "euroa"
	main()
	
	def laske_alennelta_hinta(hinta):
	bonus = 0.0
	if 100 <= hinta < 200:
	bonus =0.05
	if hinta >= 200:
	bonus =0.10
	return hinta * (100-bonus)

2	def main();
	rivi = raw_input("Anna tuotteen alkuperainen hinta.")
	hinta = float(rivi)
	alennus = 1.0
	if hinta >= 200:
	alennus = 0.9
	if hinta >= 100 and hinta <= 200:
	alennus = 0.95
	alennettu_hinta = alennus * hinta
	print "Tuotteen alennettu hinta on:", alennettu_hinta
	main()

4	def main();
	alkup_raw = raw_input("Anna alkuperainen hinta.\n");
	alkup = float(alkup_raw);
	if alkup >= 200:
	ale = 0.90
	elif alkup >= 100:
	ale = 0.95
	else:
	ale = 1.0
	alehinta = alkup*ale
	print " Tuotteen alennettu hinta on.", alehinta
	main()