Investigating high-achieving students' code-writing abilities through the SOLO taxonomy

Ayman Qahmash
Department of Computer Science
University of Warwick
A.Qahmash@warwick.ac.uk

Mike Joy
Department of Computer Science
University of Warwick
M.S.Joy@warwick.ac.uk

Adam Boddison
National Association of Special Educational Needs
Adamb@nasen.org.uk

Abstract

Computer Science Educationalists have implemented educational taxonomies which enhance the pedagogy for introductory programming modules. The SOLO taxonomy has been applied to measure students' cognitive abilities in programming by classifying students' exam answers. However, SOLO provides a generic framework that can be applied in different disciplines, including Computer Science, and this can lead to ambiguity and inconsistent classification. In this paper, we investigate high-achieving students' coding abilities and whether they tend to manifest specific SOLO categories. We address the challenges of interpreting SOLO and the limitations of code-writing problems by analysing three specific programming problems (Array Creation, Linear Search and Recursion) and solutions to those problems presented by a group of nine students. Results for the first programming problem show that six students' responses fell into the highest possible category (Multistructural) and the remaining three were categorised in the second highest category (Unistructural). For the second problem, eight students' responses fell into the Multistructural category, while only one response was categorised as Unistructural. For the third problem, two students provided Multistructural solutions and five students' solutions were Unistructural, but two further students showed a lack of understanding program constructs in their solutions, which were then categorised as Prestructural.

Keywords: programming, code-writing, SOLO

1. Introduction

Educational taxonomies have been implemented in many educational domains to enhance pedagogy, assessments and teaching methods, all of which affect students' learning, knowledge and skills. There have been many attempts to apply different taxonomies, and these have been valuable in providing insights into computer science education (CSE) to understand different educational factors. Well-developed educational taxonomies, such as Bloom, revised Bloom and SOLO (Bloom, 1956; Krathwohl, 2002; Biggs, 2014), have been applied to measuring students' outcomes as well as to classifying exam questions based on what they are supposed to measure. Although an educational taxonomy provides a generic framework that can be implemented in various disciplines, educators may not always come to a constant agreement on classifications (Fuller, 2007). In this study, SOLO has been chosen for classifying students' learning outcomes as SOLO provides a hierarchy for measuring assessments and classifying students' responses.

This paper is structured as follows. A brief background of educational taxonomies are introduced, followed by a discussion of taxonomies within the context of Computer Science and our justifications for applying an educational taxonomy are discussed. Research questions, methods, procedures and analyses are outlined in the methodology section, and finally, results are presented in the discussion section.

2. Background

The structure of the Observed Learning Outcome (SOLO) taxonomy (Biggs, 2014) aims to distinguish students' cognitive levels, which are required during their learning process. The first level is *Prestructural* (P), where a student is provided with a new problem and irrelevant information. At this stage, the student has not understood the problem and tries to use simple information to solve it. The second level is *Unistructural* (U), as the student starts to focus on one single aspect that can be used to solve the problem. The third level is *Multistructural* (M), where the student starts to understand more than one factor that may help to solve the problem. The fourth level is *Relational* (R), which focuses on the qualitative development as the student starts to understand and identify relations between several aspects. The fifth level is *Extended Abstract* (EA), where the student manifests the ability to

hypothetically think about other new factors that may help to solve the problem. In addition, the student may show the ability to generalise, evaluate and/or apply the knowledge to other problems.
Another widely used educational taxonomy is that of Bloom, which focuses on three main domains: cognitive, affective and psychomotor (Bloom, 1956). The first level is *Knowledge* which refers to a student's ability to recall basic knowledge, facts, concepts and terms, whereas the second level, *Comprehension*, describes a student's ability to understand, translate and interoperate facts and concepts. A student can demonstrate a meaningful description of a problem in their own words. The third level is *Application* which indicates that a student can apply abstract knowledge to a new problem. The fourth level is *Analysis,* where a student exhibits the ability to decompose a complicated problem into integral parts and to understand the relationships between all parts. The fifth level is *Synthesis* which describes a student's ability to compose integral elements into a new meaningful solution. The highest level is *Evaluation*, which refers to making judgements based on acquired knowledge and experience. A revised Bloom taxonomy has since been introduced by Krathwohl (2002), and which provides a two-dimensional framework consisting of knowledge and cognitive processes. The revised knowledge dimension includes an extra fourth subcategory, compared to the original taxonomy. Similar to the original taxonomy, the cognitive process dimension consists of six levels. However, the revised taxonomy renames the categories as verbs, and Synthesis swaps places with Evaluation and is renamed to be Create as shown in Fig. 1.

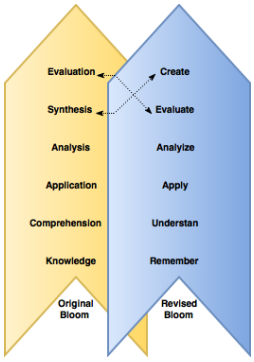

Figure 1: Original and revised Bloom's taxonomies.

2.1 Bloom in Computer Science

In the computer science field, several studies have been conducted to apply Bloom's taxonomy to curriculum design, assessment design, and student outcome measurements. Dolog (2016) used Bloom's taxonomy to revise a software engineering curriculum to meet required and desirable student skills and competencies. Johnson (2012) evaluated the assessment in a Linux course using a revised Bloom taxonomy by analysing all verbs that had been used in 10 quiz questions and 39 assignments. The results indicated that 99% of the quiz questions were about memorising, whereas 11 out of 39 assignments were categorised as knowledge (recall) and only two assignments were at the evaluation level.
Johnson and Fuller (2006) conducted a study that applied Bloom's taxonomy in order to investigate computer science students' cognitive abilities. The study encountered a problem of inconsistent categorisations, for two reasons. Firstly, determining a cognitive ability, which required the student being assessed, needs a deep understanding of how a course can be taught. Secondly, it has been claimed that applying Bloom's taxonomy to programming problems proves to be a challenging process due to insufficient frameworks and a lack of CSE knowledge on how to apply Bloom's taxonomy (Whalley et al., 2006).

2.2 SOLO in Computer Science

SOLO has been applied to computer science and education, where student performance has been assessed in a few specific aspects of programming, e.g. assisting with students' code comprehension, code writing and algorithm design. Lister et al.'s study (2006) introduced a taxonomy which provides an interpretation of how SOLO could be applied to students' answers to code comprehension problems using multiple-choice questions (MCQs). However, MCQs were not adequate to elicit responses at the Relational level. Therefore, the study was extended to analyse different types of questions in which 108 students were asked to explain a segment of code in plain English, allowing students' responses to be categorised (based on SOLO) by three academics. In addition, eight expert academics were asked to

answer the 'explain in plain English' questions in order to compare both students' and experts' responses on each SOLO level. Results showed that half of the students provided Multistructural answers, in which students were only able to explain the code line by line without indicating the purpose of the code. Meanwhile, seven out of eight experts provided answers that can be categorised at the Relational level. Later, Lister et al. (2010) applied SOLO to measure student performance in code writing, relying on Biggs (1999) verbs descriptions that are suitable for each level. In addition, Hattie and Purdie's study (1998) provides examples of how SOLO can be applied to language translation. SOLO levels can be determined by how certain phrases are interpreted rather than by translating words in isolation without understanding either the relation between the words or the context. For example, word-by-word translation, which is Unistructural, might provide meaningful translation that does not reflect the purpose of the original phrase. In the context of code-writing questions, a student may provide a direct translation of a certain program specification which does not result in correct code, whereas applying some changes to produce translation which is close to a direct specification might result in valid code. Based on Hattie and Purdie's theoretical framework, SOLO categories for code writing were proposed as shown in Table 1.

phase	SOLO category	Description
Qualitative	Extended Abstract – Extending [EA]	Uses 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 tasks (Lister et al., 2010).

Initial analyses of 30 students' code-writing answers were conducted to develop the proposed taxonomy. Students were asked to write code involving three conditional statements in which providing a direct translation for sequenced conditional statements was considered Unistructural. However, when students considered removing redundancy, solutions tended to increase on the SOLO scale, becoming Relational. The students' responses fell into only Unistructural and Multistructural. However, a second analysis of a different code-writing question was conducted for 59 students. The question related to theatre ticket sales, and was more complicated than the previous question. In this case, two students' responses were categorised as Relational. Although the proposed SOLO taxonomy provides a theoretical basis for analysing students' approaches to answering code-writing questions, it is evident in the study results that levels of questions may limit students' responses to certain SOLO categories. If a student is asked to write a program to assign a value to a variable and print out the value, it is clear that the student's response will be Unistructural — there will be no chance to provide a response at any upper level. Thus, it has been recommended that further replications of this study applied to different code-writing questions be conducted (Lister et al., 2010).

Whalley et al. (2011) proposed a refined SOLO taxonomy which overcomes previous research limitations in which mapping a very contextual code-writing question to the previous SOLO taxonomy resulted in difficulties in maintaining consistent mappings (Lister et al., 2010). In this study, a grounded theory approach had been adopted to analyse nearly 750 students' responses to three code-writing questions (Discount problem, Average calculation, and Printing a box of asterisks) in order to conduct a SOLO mapping. The mapping process started with developing empirical categories consisting of silent programming elements (SPEs) to extract program constructs, syntactical elements and code features by conducting constant coding of students' codes. Coding process allow expert computer science educators to identify silent programming elements which could emerge from students' code. Producing SPEs could be advantageous and is practical for different code-writing questions. The next stage was to extract broad features that reflect a general code quality which can appear in most code, such as code redundancy and efficiency. The extracted features can indicate the level of code abstraction

based on subjective evaluations. Finally, based on the SOLO taxonomy proposed by Lister et al. (2010), three researchers categorised students' responses to investigate whether using SPEs makes the mapping process efficient.

The study produced a refined taxonomy because an issue regarding the definition of the Multistructural level had been raised during the analysis stage. A previous definition of Multistructural indicated that a 'response represented a translation that is close to a direct translation. The code may have been reordered to make a "valid" solution' (Lister et al., 2010). However, during the analysis of the Average calculation problem, some responses managed to provide a direct translation that was a correct solution, but which could be less integrated. While the response is categorised as Multistructural, it tends to be over-categorised and should be Unistructural. Therefore, Multistructural was redefined as 'a translation that is close to a direct translation. The code may have been reordered to make a more integrated and/or valid solution.'

It is clear that Whalley et al. provide a rigorous methodology, conducting a grounded theory approach to analyse a large set of data which requires a constant coding process to produce SPEs that can be reproduced for different code-writing questions. The mapping process requires expert computer science educators who are capable of identifying multiple alternative solutions or SPEs in which common features can be extracted. Students' responses might be classified as Unistructural, which should indicate at least a single concept or SPE, whereas a Multistructural response should indicate a student's understanding of multiple concepts or SPEs, which may or may not provide an integrated solution. However, a Relational response should indicate that all concepts and SPEs have been integrated, manifesting a comprehension of the relationships between all elements and features. Computer science educators should understand that classifying students' responses is based on the level of translated specifications that are required to satisfy code implementations. In other words, the level of required specifications in a certain question affects students' response classifications but not necessarily that the classification could measure student knowledge.

It has been claimed that the mapping process used in previous research (Whalley et al.,2011; Jimoyiannis, 2013) has not been consistent in defining programming constructs at the Unistructural level. Therefore, developing the building blocks may overcome the previous research limitations in order to identify programming constructs for the Unistructural level only. The building blocks should be derived from the current course curriculum while considering the knowledge that has been acquired by students. Iterative and vector questions were analysed while applying the proposed building blocks and results showed that 44% of students' performances achieved a Relational level and 3% were at a Unistructural level.

3. Methodology

Content analysis provides a systematic approach to understand and analyse documents, transcriptions, audios and videos. Bryman (2015) defines content analysis as an approach to quantify content based on predetermined categories in which analysis procedures should be systematic and replicable. Another feature of content analysis is that can be integrated with other approaches (Bryman, 2015) such as, in our case, the SOLO taxonomy.

3.1 Research questions

- How to assess students' cognitive abilities for code-writing problems?
- Do high-achieving students tend to manifest specific SOLO categories for code-writing problems?

Data consisting of nine students' exam scripts from a level 1 programming course were selected based on the students' performance in programming and mathematics. Students proved to achieve high performance based on their grades, therefore, we were interested to analyse their responses based on the SOLO taxonomy. The programming course covers programming fundamentals, Object Oriented Programming, design, constructions, and testing, using the Java programming language. Three code-writing questions were selected, each of which included different programming constructs. We adapted Whalley et al.'s (2011) analysis approach, as shown in Fig. 2, to develop the SPE for each question, to which students' responses were coded by three independent researchers. The SPE could be identified based on syntactical elements. Then, each researcher extracted general constructs, elements and

features. Those features could be abstract and imply certain code quality. The final step was to use the developed SPEs to categorise students' responses using the SOLO taxonomy. To ensure that all researchers followed same analysis steps, the analysis procedure was developed and distributed. At the end, each researcher consolidated their findings and discussed issues that might have affected the mapping process.

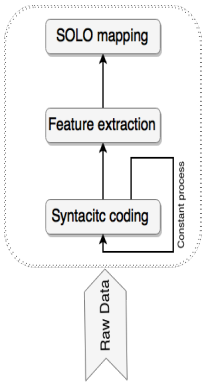

Figure 2: Bottom-up analysis approach.

3.2 Code-writing problems

3.2.1 Array Creation

The first problem was about writing a method that takes a single integer, n, as an argument to create an array of size n with random values between 0 and 100. Students should write valid code to demonstrate their knowledge of array declaration, initialisation and iteration. In addition, the code can be implemented using Java built-in Math or Random objects and their functions to generate random values to be stored in the array. We assumed that those objects had been introduced to the students in the course. However, the question included a non-direct translation of the specifications as the array must include values between 0 and 100 inclusive. In this case, Math objects have a Random method that returns a double value, greater than or equal to 0.0 and less than 1.0, which needs to be multiplied by 101 and converted to integer. Thus, the array will include values from 0 to 100. We decided that the question should be categorised Multistructural. Sample of Student solution are shown in Table 3.

N	Code

1	`public static int[] createArray(int n){`
	`  int[] newArray = new int[n];`
	`  for(int i=0;i<n;i++){`
	`  newArray[i] = (int)(Math.random() * 101);`
	`  }`
	`  return newArray;`
	`}`

Table 2: Sample of student solution for Array Creation problem.

Coding the students' answers is the next step to develop the SPEs derived from students' program code to identify program constructs and syntaxes that be used to implement such a program to solve a problem. As shown in Table 4, the main program constructs consist of method declaration, array declaration, iteration, initialization with random values, and a return statement. In most solutions, the methods were declared correctly to return the created array. However, some methods were declared to be 'static', which was not required in the question specifications. For the array declaration, all solutions declared the array with size n in a one line statement, which is more efficient than using two line statements. All solutions implemented the array iterations using one finite loop whereas there were two options to generate random values to be stored in the array. Both Math and Random Java objects were implemented, however some solutions were not able to generate random values between 0 and 100 including 100 as specified in the question.

Construct	Element	Feature
Method declaration	`Public int[] array (int n)`	Typical
	`Public void array (int n)`	Void method
Array declaration	`int [] array = new int[n];`	Efficient
Array iteration	1x for loop	Finite loop
Random value generation	Using Math object	Inclusive range
		Exclusive range
	Using Random object	Exclusive range
Return statement	Return array;	Included
		Missing

Table 3: Program constructs and features for Array Creation problem.

Based on the resulting SPEs and features extracted from students' codes, Table 5 shows the SOLO mapping.

Construct	Feature	Solutions by Student's number								
		55	14	36	91	98	78	42	79	49
Method declaration	Typical	x	x	x	x	x	x	x	x	
	Void method									x
Array declaration	Efficient	x	x	x	x	x	x	x	x	x
Array iteration	Finite loop	x	x	x	x	x	x	x	x	x
Random value generation	Inclusive range	x		x	x					
	Exclusive range		x			x	x	x	x	
Return statement	Included	x	x	x	x	x	x	x		
	Missing								x	x
SOLO mapping (1st researcher)		M	U	M	M	U	U	U	U	U
SOLO mapping (2nd researcher)		M	U	M	M	U	U	U	U	U
SOLO mapping (3rd researcher)		R	U	R	M	U	U	U	U	U
Final and agreed SOLO mapping		M	U	M	M	U	U	U	U	U

Table 4: SOLO mapping for Array Creation problem.

3.2.2 Linear Search

The second problem was to write a method that takes an array and an argument, s, as arguments, and performs a linear search on the array finding the index when s is found or returning -1 if s is not found. We agreed that the question's specification can be translated directly and should be categorised as Multistructural.

All students demonstrated a clear understanding of the question and produced code that included main constructs. As shown in Table 6 in code number 2, the student's code tended to have a redundant declared variable to be returned, thus we consider it as redundancy in the return statement.

Different constructs extracted from students' code included method declaration, array iteration, selection and the return statement. Students' solutions were then categorised based on derived features as shown in Table 8.

N	Code

1	`public int linSearch(int[] searchArray, int s){`
	`  for(int i=0;i<searchArray.length();i++){`
	`  if(searchArray[i] == s){`
	`  return i`
	`  }`
	`  }`
	`  return -1;`
	`}`

Table 5: Sample of student solution for Linear Search problem.

Construct	Element	Feature
Method declaration	`public int linearArray(int [] array, int s)`	Typical
	`public void linearArray(int [] array, int s)`	Void method
Array iteration	`for(int i=0;i<s;i++)`	Finite loop
Selection	`If statement`	Valid condition
Return statement	`int find = -1;` `Return find;`	Redundant
		non-redundant

Table 6: Constructs and features for Linear Search problem.

Construct	Feature	Solutions by Student's number								
		78	98	91	79	14	49	42	36	55
Method declaration	Typical	x	x	x	x	x	x	x	x	
	Void method									x
Array iteration	Finite loop	x	x	x	x	x	x	x	x	x
Selection	Valid condition	x	x	x	x	x	x	x	x	x
Return statement	Redundant								x	x
	non-redundant	x	x	x	x	x	x	x		
SOLO mapping (1st researcher)		M	M	M	M	M	M	M	U	U
SOLO mapping (2nd researcher)		M	M	M	M	M	M	M	M	M
SOLO mapping (3rd researcher)		M	M	M	U	M	M	M	M	U
Final and agreed SOLO mapping		M	M	M	M	M	M	M	M	U

Table 7: SOLO mapping for Linear Search problem.

3.2.3 Recursive method

The third question was about writing a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between A[0] and A[n-1], A[1] and A[n-2], etc.). The question aimed to measure a student's ability to implement a recursive method, which is considered to be a difficult concept to be understood by novice programmers. Thus, we agreed to categorise this question to be Rational, as the question included additional complex constructs along with applying the recursion concept. A typical solution passes to the method an array together with a variable that keeps track of the array index that traverses incrementally from left to right. Then, it is important to have a second variable which keeps track of the array index that traverses in the opposite way. In addition, edges of the array must be checked, in order to calculate differences between the edges. Table 9 shows student code which meets the question's specifications and is considered to be valid code, and Table 10 shows constructs, elements and features extracted from students' code. Most important constructs which differentiate students' solutions for the SOLO mapping are edges, difference calculation and recursive method invocation. Given the fact that the nature of recursion involves a degree of abstraction, novice students encounter difficulties implementing recursive methods (Wirth, 2014). Therefore, students' solutions manifest different levels of SOLO categories ranging from the lowest to the highest (which is Rational in this question). Two students were not able to understand the question requirements and provided solutions lacking constructs related to the question. Table 11 shows students' SOLO categorisations.

N	Code

1	`public int oppPairs(int[] array, int pos){`
	`  int pos2 = array.length() - 1 - pos;`
	`  if(pos2 < pos)`
	`  return 0;`
	`  else`
	`  return array[pos2] - array[pos] + oppPairs(array, ++pos);`
	`}`

Table 8: Sample of student solution Recursive problem

Construct	Element	Feature
Method declaration	`Public int oppPairs(int [] array, int pos)`	Typical
	`Public int oppPairs(int [] array)`	Missing argument
	`Public void oppPairs(int [] array, int pos)`	Void method
Variable assignment	`int pos2=array.length() -1-pos;`	Efficient
edges	`If (pos2<pos)`	Valid
Difference calculation	`int diff = array[pos2]-array[pos] + array[pos2-j]-array[pos+i];`	Invalid
	`int diff = array[pos2]-array[pos] + oppPairs(array,++pos);`	Efficient
recursive invocation	`oppPairs(array,++pos)`	Valid argument
		invalid argument
Return statement	`Return array;`	non-redundant

Table 9: Constructs and features for Recursive problem.

Construct	Feature	Solutions by Student's number								
		49	14	79	91	98	78	55	36	42
Method declaration	Typical			x		x	x		x	x
	Void method	x			x			x		
	Missing argument		x							
Variable assignment	Efficient							x		x
edges	Valid									x
	invalid								x	
Difference calculation	efficient								x	x
recursive invocation	Valid argument				x	x	x	x	x	x
	Invalid argument			x						
Return statement	Non-redundant		x	x	x	x	x	x	x	x
SOLO mapping (1st researcher)		P	P	U	U	U	U	U	R	R
SOLO mapping (2nd researcher)		P	P	M	M	M	P	U	R	R
SOLO mapping (3rd researcher)		U	U	U	U	U	U	U	M	R
Final and agreed SOLO mapping		P	P	U	U	U	U	U	R	R

Table 10: SOLO mapping for Recursive problem.

4. Discussion

Despite the effort applied to developing a SOLO taxonomy for code-writing questions, mapping students' responses based on a specific SOLO taxonomy has a degree of ambiguity and inconsistency. SPEs had therefore been introduced by Whalley et al. (2011) to minimise the mapping ambiguity and inconsistency. In addition, limitations of code-writing questions affect the mapping of students' responses as certain types of question do not allow for high order thinking to be manifested in the students' code. For example, if the question tends to measure student knowledge on how to declare a variable and assign a value to the variable, the student makes direct translations of what is required. Clearly, the student's code can not be categorised EA as the question is limited to specific requirements. We find that identifying program constructs and extracting the quality features allow more constant mapping provided by all researchers, and we held a consolidation meeting to refine extracting program constructs and features for question three. In addition, we evaluated the importance and the weight of certain constructs that might have affected the mapping process. For example, there was a concern raised by one researcher questioning method declaration using the modifier static, and thus the mapping had to be slightly changed. For instance, in Table 11, student 36 manifested three main constructs (edges, difference calculation and recursion invocation) and had a slight error while checking the edges, and the student's response categorised Relational the same as responses that manifested all three constructs. The student's response that was categorised Relational should manifest all main constructs and features showing understanding of the relationship between them (Whalley, 2011).

Another challenge was the choice of the questions as we had been limited to only three code-writing questions that had been included in the exam script. Limitations of questions prevented students' ability to be manifested and categorised in a higher category. The three questions had been categorised Multistructural, Multistructural and Relational respectively, thus those categories represented the highest categories for each question. In addition, we agreed to consider SOLO categories for each question when mapping students' responses, so if a student's response had been categorised higher than the question level, the category should be degraded. We agreed to categorise the questions based on the level of translations and concepts needed to be measured. Therefore, mapping students' responses for code-writing questions should be accorded to the level of translations of required specifications in the code-writing questions (Whalley, 2011).

Our aim was to investigate high-performing students' responses according to the SOLO taxonomy. Despite the limitations and challenges addressed earlier, results show that high-performing student manifest the ability to understand code-writing problems and provide solutions that might be categorized at the highest possible SOLO category. In question one (Array Creation), six students' responses fell into the highest possible category whereas the rest of students' responses were categorised in the second highest category (Fig. 3). In question two (Linear Search), eight students' responses resided in the Multistructural category, which is the highest category for question two, while one

responses were categorised as Unistructural as shown in Fig. 4. As we mentioned earlier, question three focused on recursion which is one of the most difficult concepts for novice programmers. Therefore, question three was categorised as Relational where students' responses might manifest a degree of abstraction that might vary from one student to another in which responses could be categorised at different levels. Fig. 5 shows that two students provided solutions categorized at the highest possible level where five students' solutions manifested direct translations with invalid solutions categorized as Unistructural. Two students showed a lack of understanding program constructs in their solutions which had been categorised as Prestructural.

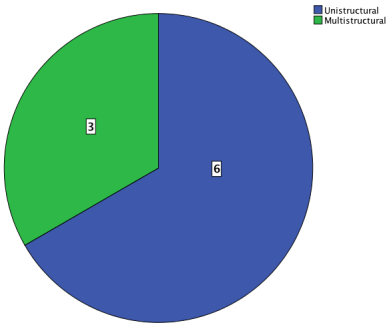

Figure 3: Students numbers mapped into SOLO for Array Creation problem.

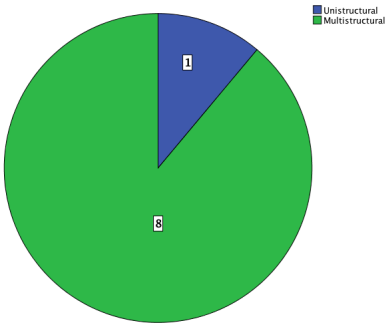

Figure 4: Students numbers mapped into SOLO for linear search problem.

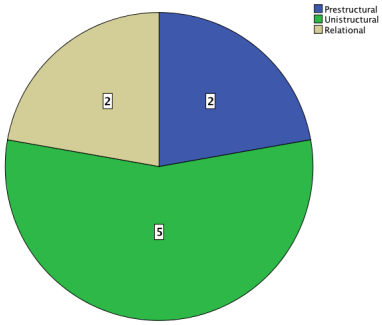

Figure 5: Students numbers mapped into SOLO for Recursive problem.

5. Conclusion

Educational taxonomies provide a framework for CSE to categorise students' cognitive abilities in the computer science field. Several attempts have been made to apply Bloom's taxonomy to categorise student code, but have resulted in a great deal of ambiguity as Bloom does not provide descriptions that can be interpreted easily in computer science. However, the SOLO taxonomy has been applied to classify students' codes and algorithm designs. In this paper, we have adapted Whalley et al.'s (2011) framework, which has allowed us to code students' responses for code-writing questions and to develop SPE and quality features which have assisted us to categorise students' responses. Including the first author, two researchers have replicated the analysis procedures to ensure that analysis has yielded consistent results. The number of high-achieving students' responses were categorised at the highest possible level for two of the three questions which were analysed, although only two students' responses were categorised at the second highest and highest levels for the remaining question (which focused on the complex concept of recursion).

References

Anderson, L. W., Krathwohl, D. R., Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., & Wittrock, M. (2001). A taxonomy for learning, teaching and assessing: A revision of Bloom's taxonomy. New York. *Longman Publishing*.

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

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Bloom, B. S., Engelhart, M. D., Furst, E. J., Hill, W. H., & Krathwohl, D. R. (1956). Taxonomy of educational objectives, handbook I: The cognitive domain.

Bryman, A. (2015). *Social research methods*. Oxford university press.

Dolog, P., Thomsen, L. L., & Thomsen, B. (2016). Assessing Problem-Based Learning in a Software Engineering Curriculum Using Bloom's Taxonomy and the IEEE Software Engineering Body of Knowledge. *ACM Transactions on Computing Education (TOCE)*, 16(3), 9.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. In *ACM SIGCSE Bulletin (Vol. 39, No. 4, pp. 152-170)*. ACM.

Ginat, D., & Menashe, E. (2015). SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 452-457)*. ACM.

Hattie, J.A. and Purdie, N. 1998. The power of the solo model to address fundamental measurement issues. *In Teaching and Learning in Higher Education*, Edited by: Dart, B. and Boultonlewis, G. Victoria, Australia: ACER.

Jimoyiannis, A. (2013). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education*, 4(2), 53-74.

Johnson, C. G., & Fuller, U. (2006). Is Bloom's taxonomy appropriate for computer science?. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling* 2006 (pp. 120-123). ACM.

Johnson, G., Gaspar, A., Boyer, N., Bennett, C., & Armitage, W. (2012). Applying the revised Bloom's taxonomy of the cognitive domain to linux system administration assessments. *Journal of Computing Sciences in Colleges*, 28(2), 238-247.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into practice*, 41(4), 212-218.

Lister, R., Clear, T., Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., ... & Thompson, E. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, 41(4), 156-173.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, 38(3), 118-122.

Wirth, M. (2014). The Canny Skipper-A Puzzle For Demonstrating Data Structures And Recursion. In *Proceedings of the Western Canadian Conference on Computing Education* (p. 16). ACM.

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume* 114 (pp. 37-46). Australian Computer Society, Inc..

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52* (pp. 243-252). Australian Computer Society, Inc..