Revisiting models of human conceptualisation in the context of a programming examination

Jacqueline Whalley and Nadia Kasto
School of Computing and Mathematical Sciences
AUT University
PO Box 92006, Auckland 1142, New Zealand
{nkasto,jwhalley}@aut.ac.nz

Abstract

This paper reports on an evaluation of the Block model for the measurement of code comprehension questions in a first semester programming examination. A set of exam questions is classified using the Block model and two commonly employed taxonomies, SOLO and Bloom. We found that some of the problems inherent in the application of Bloom and SOLO taxonomies also exist in the Block model. Some of the difficulties associated with SOLO and Bloom's taxonomy are due to the wide breadth of the dimensions. These difficulties are to some degree mitigated by the limited breadth of the Block model dimensions and we found that the Block model provided a better way of describing novice programming code comprehension tasks because of the increased granularity that it provides.

Keywords: code comprehension, novice programmers, Block model, SOLO, Bloom's taxonomy.

1 Introduction

Teachers of computer programming have experienced difficulty in judging the cognitive complexity of learning tasks and test items. A relatively accurate and simple way is required for determining the difficulty inherent in our teaching and assessment programs: *"...we as educators are continually underestimating the difficulty of the tasks that we are asking students to undertake*" (Whalley, Clear and Lister 2007).

Computer science educators have attempted to apply models and taxonomies of human conceptualisation to aspects of the teaching and learning of computer programming with varying degrees of success. The most widely adopted taxonomies to date have been the Bloom (Bloom 1956) and SOLO (Biggs and Collis 1982) taxonomies. Recently a new taxonomy has been developed specifically for application to the design of tasks for computer programming. This paper reports on an investigation of the use of that model to determine the difficulty of a set of test questions.

2 Background

In 1956, Bloom produced a taxonomy that consisted of a hierarchy of learning objectives ranked according to their expected cognitive complexity (Figure 1). The taxonomy is a behavioural classification system of educational objectives. Many variants of the taxonomy have been proposed but the most widely accepted (Figure 1) is the revised Bloom's taxonomy (Anderson et al. 2001). This version of the taxonomy adds a knowledge dimension, which specifies the type of information that is processed, to a revised version of the original cognitive process dimension. Traditionally a strict inclusive hierarchy has been assumed for the cognitive process dimension where each category is assumed to include lower ones.

Bloom	Revised Bloom
Evaluation	Create
Synthesis	Evaluate
Analysis	Analyse
Application	Apply
Comprehension	Understand
Knowledge	Remember

Figure 1: The cognitive process dimension; (left) Bloom's and, (right) revised Bloom's taxonomy

Bloom's taxonomy has been applied to computer science for course design and evaluation (Scott 2003), structuring assessments (Lister and Leaney 2003, Lister 2001), specifying learning outcomes (Starr, Manaris and Stavely 2008) and comparing the cognitive difficulty of computer science courses (Oliver et al. 2004).

The revised and the original Bloom's taxonomy have been used in attempts to improve the instruction and assessment of programming courses (e.g., Abran et al. 2004, Shneider and Gladkikh, 2006, Thompson et al. 2008, Khairuddin and Hashim 2008, Alaoutinen and Smolander 2010, Whalley et al. 2006, Whalley et al. 2007, Shuhidan, Hamilton and D'Souza 2009).

The use and interpretation of Bloom and the revised Bloom's taxonomy for describing computer science tasks has been found to be problematic (Fuller et al. 2007, Thompson et al. 2008, Shuhidan, Hamilton and D'Souza 2009, Meerbaum-Salant, Armoni and Ben-Ari 2010). Much of the research shows that it can be difficult to reach a consensus on an interpretation for the computer programming education domain (Johnson and Fuller 2006). In a recent study Gluga et al. (2012) confirmed that many of the ambiguities in the application of Bloom's taxonomy to the assessment of computer programming are due to the necessity to have a deep understanding of the learning context in order to achieve an accurate classification. They also noted that the classifiers often had preconceived misunderstandings of the categories and differing views on the complexity of

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136. A. Carbone and J. Whalley, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

tasks and the sophistication of the cognitive processes required to solve them. This may be due to the difficulty that the educators have remembering the cognitive complexity of such a task when they were learning to program. A much higher cognitive load exists for a novice programmer writing a simple function than for an experienced programmer. Additionally it has been reported that the ordering of cognitive tasks in Bloom's taxonomy does not readily map to the learning trajectories of many novice programmers (Lahtinen 2007).

As a result of these difficulties, several variants of Bloom's taxonomy have been proposed specifically for computer programming education (e.g., Schneider and Gladkikh 2006, Fuller et al. 2007, Bower 2008). These variants have not been widely adopted by computer science educators and researchers. Perhaps this is partially due to the fact that the appropriateness of Bloom's taxonomy for the design of learning activities and assessments has been disputed. The presupposition that there is a necessary relationship between the questions asked and the responses elicited is not a valid one because a question could potentially elicit responses at different levels (Hattie and Purdie 1998).

Biggs and Collis (1982) surmised that Bloom levels reflect a teacher imposed view of what it means to have achieved full mastery whereas SOLO levels come from an understanding of the student learning process. The focus of Bloom is to assist in the development of educational objectives, while the SOLO taxonomy focuses on the cognitive process used to solve problems. SOLO, unlike Bloom, does not assume a relationship between the task and the outcome so outcomes to a specific task may be at different levels for different students. Additionally, while Bloom separates knowledge from the intellectual processes that operate on this 'knowledge', the SOLO taxonomy is primarily based on the processes of understanding used by the students when solving problems. Therefore, knowledge is inferred in all levels of the SOLO taxonomy. It may be due to these differences that educators and researchers have had greater success in using SOLO to describe programming tasks (code comprehension and code writing), to classify student responses to those tasks and to gain some insight into the students' cognitive processes (e.g., Lister et al., 2006, Philpott, Robbins and Whalley 2007, Sheard et al. 2008, Clear et al. 2008).

Both taxonomies have been used, independently, to analyse the same set of programming assessment questions and responses (Whalley et al. 2006). Thompson et al. (2008) noted that the Bloom category for a programming task can be meaningfully mapped to a number of categories in the SOLO taxonomy and that a combined version of these taxonomies may provide a richer model with which to design and describe programming tasks. Inspired by Thompson's observation a hybrid taxonomy was proposed that combines aspects of the Bloom and SOLO taxonomies (Meerbaum-Salant, Armoni and Ben-Ari 2010). The combined taxonomy consists of the SOLO categories of *unistructural*, *multistructural* and *relational* and three Bloom categories *understand* (U), *apply* (A) and *create* (C). The taxonomy was structured so that the three SOLO levels formed super-categories each containing the three Bloom levels as subcategories (Figure 2) . This taxonomy was then used to analyse the correlation between student performance on a task and the relative complexity of the task as defined by the classification of the task using the combined taxonomy. The authors believe that their "*findings suggest that the combined taxonomy captures the cognitive characteristics of CS practice*". They also recommend this integration of taxonomies as a research framework that is applicable to the specific needs of CS education research. However, they also note that the taxonomy requires further investigation and validation. To date this work has not been reported in the literature.

Unistructural			Multistructural			Relational		
U	A	C	A	A	C	U	A	C

Figure 2: The combined taxonomy

The Block model (Schulte 2008) is an educational model of program comprehension. It is structured as a table consisting of three knowledge dimensions and four hierarchical levels of comprehension. The table consists of 12 blocks (cells) and each block is designed to highlight one aspect of the program comprehension process (Figure 3). The conceptualisation of the hierarchical levels takes inspiration from Kintsch's expanded text comprehension theory (1998).

Macro structure	Understanding the overall structure of the program text	Understanding the "algorithm" of the program	Understanding the goal / the purpose of the program in its context
Relations	References between blocks (e.g.: method calls, object creation, accessing data	Sequence of method calls	Understanding how subgoals are related to goals, how function is achieved by subfunctions
Blocks	Regions of interest (ROI) that syntactically or semantically build a unit	Operation of a Block, a method or ROI (as a sequence of statements)	Function of block, maybe seen as a subgoal.
Atoms	Language elements	Operation of a statement	Function of a statement. For which goal is only understandable in context
	***T*ext surface**	***P*rogram execution** (data flow and control flow)	***F*unctions** (as means or as purpose, goals of the program)
	Structure		***Function***

Figure 3: The Block model (Schulte 2008)

Question	Type	Revised Bloom's cognitive dimension		Block Model level	Block Model comprehension dimension	SOLO	% correct answers
2	Basics	Remember		Atom	Text surface	U	50%
4	Syntactic errors	A	Remember	Atom	Text surface	U	68%
		B	Understand	Block	Text surface	M	53%
		C	Understand	Relations	Text surface	M	31%
7A	Tracing	Apply		Block	Execution	M	77%
7B	Tracing	Apply		Block	Execution	M	64%
7C	Tracing (with selection)	Apply		Block	Execution	M	81%
7D	Tracing (with iteration)	Apply		Relations	Execution	M	22%
7E	Tracing (with iteration)	Apply		Relations	Execution	M	27%
5	Skeleton Code	Analyse		Relations	Functions	R	42%
6	Parsons Puzzle	Apply		Block	Functions	M	60%
10A	Code Intent	Understand		Macro	Functions	R	36%
10B	Code Intent	Understand		Macro	Functions	R	9%
10C	Code Intent	Understand		Macro	Functions	R	6%

Table 1: Classification of exam questions

The intention behind the Block model's development was to provide a relatively simple model, compared with other existing taxonomies and models, to support research into the teaching of computer programming. The model was evaluated as a tool for the planning and evaluation of lessons about algorithm design (Schulte 2008). It was found that the Block model was simple, constructive and communicative. However the model has not yet been used as a framework for research into the teaching and learning of computer programming.

In a recent paper the Block model was used to map a variety of selected models of program comprehension in order to assist in the conceptualisation of those models (Schulte et al. 2010). As a result of this comparative analysis of models the authors suggest that the process of knowledge acquisition by novice programmers described in terms of the Block model might be represented as a holey patchwork quilt and that the Block model might help us identify what holes (empty cells) exist and why a student's knowledge is "fragile". We were interested in investigating the usefulness of the Block model for measuring and evaluating programming tasks and also for investigating the cognitive processes employed by students to solve the problems.

In this preliminary study we employed a set of programming comprehension questions, from a first semester programming examination, in order to analyse the similarities and differences of the Block model with other models.

3 Analysis and Discussion

What follows is a discussion of the analysis of a small set of program comprehension questions, given in the same pen and paper examination, collated by question type. Table 1 gives an overview of the classification of the questions. The revised Bloom classification was carried out using the vignettes and principles described by Thompson et al. (2008) and Whalley et al. (2006) . In accordance with this set of guidelines we classified the cognitive process dimension at the category level rather than the sub category level. In classifying the questions using the SOLO taxonomy we applied the principles and guidelines provided by Biggs and Collis (1982) and from the SOLO categories established by the BRACElet project for 'code intent' comprehension tasks (Clear et al. 2008) . The Block model classification was carried out using the cell descriptions shown in Figure 3.

One challenge we faced was in determining exactly what an atom or a block is. It could be argued that this is dependent on the stage of development that the individual learner has reached. This observation has been previously made with respect to salient elements in novice programming tasks (Whalley et al., 2010). Here in assigning classifications we have assumed a norm for all students based on our experiences in teaching novice programmers. We have taken the notion of an *atom* to be the simplest salient element (for example a variable declaration and assignment) and a *block* to be a single method, loop or selection statement. Therefore at the *relations* level we consider a relationship to be a reference between blocks or between a block and an atom.

For each question the student performance on that question was also analysed. In our analysis we are interested in what skills, knowledge base and cognitive processes are required to successfully answer the question. Finally we then compared the actual relative difficulty of the questions in the context of the examination (as indicated by the percentage of students who gave a fully correct answer) with the levels of difficulty of those questions as indicated by the taxonomies and models.

Question 2: Matching Terms to Code

Question 2 presented students with a class definition that had ten lines of code underlined and each annotated with a letter. Students were asked to match 7 definitions to the appropriate line of code. This question required students to recall factual knowledge and was classified as *remember*. In terms of SOLO this question requires students to focus on a single language construct and is therefore a *unistructural* question.

Because students are focusing on a single language element this question is considered to be at the atom level

in the Block model. The *text surface* dimension of the Block model is associated with the external representation of the program, "*it is the code a person reads in order to comprehend the program*" (Schulte et al. 2010). In order to answer question 2 the students do not need to go beyond understanding the rules of discourse (grammar) of the program code. They certainly do not need to understand or have knowledge of the data and control flow or goal of the atom of code in order to answer this question correctly. Therefore this question is an *atom* level, *text surface* question.

Question 4: Syntax Errors

In question 4 (see Appendix) students were asked to find 8 of 11 syntax errors in a complete class. The type of syntax error had a great affect on the number of students who were able to correctly locate and identify the error. It seems, not unexpectedly, that the difficulty of the task (as measured by student performance on the task) is related to the type of knowledge that is required which in turn is directly related to the type of bug or error to be identified. We found that when we mapped each error identification question to the Block model clear groupings emerged based on the level of comprehension required to reach the correct answer. The lowest level of syntactic errors, which we grouped together as 4A, consisted of errors such as missing semicolons, a missing bracket in a method declaration and typographical errors such as Return rather than the correct r eturn keyword. All of these errors can be found without reference to the rest of the program structure. They focus on a language element and therefore with respect to the level of student program comprehension required to answer the question they can be classified as *text surface* at the *atom* level. Identifying these types of errors can also be considered to be a *unistructural* task and in terms of Bloom they require the students to *recognise* an error that they would have seen repeatedly during their course of study.

The syntactic errors that we grouped as question 4B consist of mismatches either between the return data type of a method and the data type of the value returned or between a parameter identifier in the method declaration and the identifier used to represent that method parameter in the method body. These errors are all located within a block of code and consist of a sequence of atoms. One error was positioned in a selection statement. In order to locate these errors the students must understand the syntactic structure of the block so these error identification tasks were classified as requiring *text surface* knowledge at the *block* level. It is not necessary to operate at the *relations* level in order to identify these errors. We classified the 4B errors to the SOLO *multistructural* category because they focus on more than one language construct but to answer correctly them does not require the students to understand the relationship between the constructs and the problem can be solved by knowing the required structure of the code rather than the purpose or goal of the code. In order to identify this type of error students must not only recall basic syntax rules but also identify where there is an incorrect application of the rule. In order to do this the students must *understand* (Bloom's category) the rule.

Finally, syntactic errors that were grouped together as 4C consisted of bugs such as an incorrect method call or a data type mismatch for a global variable. In order to recognize this type of error, the students need to be aware of the relationships between various blocks in the code and therefore required comprehension of *relations*. In order to identify these bugs the students are still operating at the *text surface* where an understanding does not need to extend beyond the application of their knowledge of the 'grammar' of the code.

While the different forms of question (4A, 4B, 4C) were classified into three separate categories in the Block model they were classified into only two different categories when Bloom and SOLO classifications were applied. The Block model was the only classification system to put the three different types of questions into separate categories.

Question 7: Code Tracing

Tracing questions are solved by tracking data through the code line by line. This question type has not been previously classified using the SOLO taxonomy. However, in a study that analysed student answers to 'code intent' questions it was noted that "*a student may hand execute code and arrive at a ...* [correct]*... final value but ... the student may not manifest an understanding of what the code does*". Such student responses were classified as *multistructural* (Lister et al. 2006). Extrapolating this to tracing questions it is clear that it is not necessary to understand the purpose of the code to reach the correct answer and question 7 A-E should be classified as *multistructural*.

These questions require the students to apply a known process or strategy and are therefore classified as *apply* in the revised Bloom's cognitive dimension.

The students need to have knowledge of the data flow and in some cases control flow of a simple Java method in order to answer code tracing questions. In order to operate at the *program execution* level they must also operate at the lower *text surface* level. They do not need to extend to the *functions* domain of the Block model. Therefore, the tracing questions in this exam are all posed within the *program execution* knowledge dimension of the Block model.

The aspect in which these questions differ is the level of the task when classifying the questions using the Block model. The Block model was the only system that differentiated amongst these tracing questions. The questions were classified into two different levels within the Block model. For these questions code that contained iteration were classified at the *relations* level whereas code without iteration were classified as *block* level questions.

Question 5: Skeleton Code (with scaffolding)

The skeleton code for question 5 is a class definition, taking up a page and a half, containing two private data members (one of which is an ArrayList), a single constructor, and three methods, which add to, delete from, and print the contents of the ArrayList. After the code, the students are set the following task for refactoring the code: "The table below shows the missing lines of code, but not necessarily in the correct order. It also has one extra line of code that is not needed. Identify

which line of code should go where …" In-line comments are provided as a scaffold to help the students identify the appropriate lines of code. The scaffolding means that it is not necessary for the students to identify the overall goal of the missing lines and the blocks in which the line must be placed because this is provided. It does still, however, require them to understand the various relationships between lines of code in order for them to select the correct missing line. For example see Figure 4 where there is a need to understand the connections between fields as parameters to an external constructor method for a Lot object and the Lot object and an ArrayList method call as well as the sub-goals of these method calls. Question 5 was therefore classified as a SOLO *relational* question and in the *relations-functions* of the Block model. This question requires the students to differentiate the relevant from the irrelevant lines of code and to focus on the sections of code within the class that are relevant to the differentiation task. Therefore this question was classified as *analyse*. A similar skeleton code question has been reported previously and was also classified at the *analyse* level of Blooms cognitive process dimension (Whalley et al. 2006).

```
private ArrayList<Lot> lstLots;
private int nextLotNumber;
....
/**
 * A simple model of an auction
 * @author David J. Barnes and Michael
 Kolling */
public void enterLot(String description)
{
   //create new Lot
   Lot lot = new Lot(nextLotNum,description);
   //store it in the ArrayList
   <missing code>
   nextLotNumber++;
}
```

Figure 4: Part of question 5 code (adapted from Barnes and Kölling 2006)

Question 6: Parsons puzzle (with structure)

Question 6 is a Parsons puzzle (Parsons and Haden 2006) where students are presented with jumbled lines of code for a Java method (see Appendix). They are provided with the purpose of the method which is to count the number of occurrences of a character in a string and a structure for the method defined by a set of nested braces and blank lines. The students are required to place the lines of code into the correct order.

Classifying this question using SOLO is difficult. It could be argued that even though the students are provided with the overall purpose of the code they still have to understand the code as a whole in order to reach the correct answer. Therefore it should be considered to be a *relational* question. Additionally if we take this viewpoint the revised Bloom's cognitive level of the question must be *analyse* because the students are determining how the lines of code fit within the overall structure and purpose of the code. However research using these puzzles points to the fact that students typically *apply* a set of heuristics to solve the problem (Denny et al. 2008). For example, the final line of the method must be the return statement and the first line the method header. Even in determining the position of the loop in relation to the selection statement the variable *i* is defined in the loop and then used in the 'if statement'. Understanding the relationship between the two lines of code and the variable *i* can be seen as applying a more sophisticated heuristic. On the other hand it could be seen as manifesting an understanding of the purpose of the variable *i*. Either way in terms of SOLO, the question is *multistructural* because although connections between parts of the code must be made, the question does not require meta-connections to be made.

The Parsons puzzle examined here is classified at the *apply* level, in the revised Bloom's cognitive dimension, because it is possible to solve this problem correctly by applying known heuristics.

This question requires students to operate at the *block* level. It is not necessary for them to be able to understand the connections between the blocks to solve the problem because they can use heuristics. Although the students are given the overall goal of the method it is possible to solve this Parsons puzzle without understanding the overall goal. However it seems that the students must at least understand the sub-goals of the constituent blocks and atoms in order to solve this puzzle correctly. Therefore we have classified this puzzle in the *functions* knowledge domain. It should be noted that a more complex puzzle, without scaffolding, may require a deeper understanding of the logic and flow of the algorithm and be at the *relations* level of the *functions* domain. Therefore, unlike tracing questions, we cannot claim that all Parsons puzzles have a predetermined classification.

Question 10: Code Intent

Questions 10 A, B and C all required the students to explain the purpose or goal of a single method. It is clear that such a question moves beyond the structure of the code, data flow and control flow, and is within the Function cognitive dimension of the Block model. As an example we will consider question 10B (Figure 5). To solve this question the students need to understand the connections between the three blocks of code in order to infer an overall purpose.

```
public void method10B(int iNum)
{                                                    Block 1
   for(int iX = 0; iX < iNum; iX++)  Block 2
   {
      for(int iY = 0; iY < iNum; iY++)
      {
         System.out.print("*");     Block 3
      }
   }  System.out.println();
}
```

Figure 5: Question 10B and its constituent blocks

There are a number of possible ways that a student might solve this problem. They might apply a top-down comprehension strategy by identifying the sub-goal of *Block 3* before trying to understand the function of *Block 2*. When considering *Block 2* they see *Block 3* as an atom (which prints a line of iNum stars) and that *Block 2* executes *Block 3* followed by a carriage return iNum times. The outer block might be processed in a similar way in order to arrive at the purpose of the method. In this way it can be argued that a student is reaching an

understanding of the relationship between the three blocks and then inferring an overall purpose. The same outcome might be achieved by tracing the code, a bottom-up strategy, in order to try to see the relationships between the blocks and arrive at a conclusion as to the purpose of the code. Or they may apply a combination of both. Regardless of the strategy they apply, to reach a correct answer the student is operating at the highest level within the *functions* dimension because an "understanding the goal or function of the program" is required. 'Code intent' questions, similar to the ones in this examination have been consistently classified as SOLO *relational* (see Clear et. al 2008) and this gives weight to our classification of questions 10A-C as *relational*. In past work 'code intent' questions have been classified as *understand* (Thompson et al. 2008, Whalley et al. 2006). Our initial classification was at this level. However we believe that the cognitive processes used by novice programmers when trying to solve 'code intent' questions are more complex than previously assumed. A fuller discussion of this aspect of using Blooms taxonomy to classify program comprehension tasks is provided in the next section.

3.1 Using Bloom's taxonomy

Like many educators in science disciplines we have found it difficult to apply the Bloom and the revised Bloom taxonomies. There have been several studies that indicate that the order of the levels changes depending on the task. For example in a test on atomic structure it was found that *synthesis* and *evaluation* were placed between *knowledge* and *comprehension*. A test related to glaciers found that *synthesis* lay between *knowledge* and *comprehension* (Kropp and Stocker 1966). Similarly, we believe that the cognitive dimension hierarchy does not map comfortably with computer programming tasks.

In classifying 'code intent' questions Thompson et al.'s (2008) revised Bloom vignettes and definitions indicate that this type of question is at the *understand* level. In the revised Bloom's taxonomy *understand* is defined as '*constructing meaning from instructional messages*' which is interpreted by Thompson et al. (2008) to include "*explaining a concept or an algorithm or design pattern*". Tracing questions were classified, by Thompson et al. (2008) at the higher revised Bloom level of *apply*. *Apply* is defined as "*carrying out or using a procedure in a given situation*" and clearly hand execution of code is a process which students must apply in order to answer a code tracing question.

Past research has shown us that novice programmers find 'code intent' questions more difficult than tracing questions and Parsons puzzles (Lopez et al. 2008). A study which examined the approaches of experts vs. novices in solving these types of problems also illustrated that there is likely to be a higher cognitive load and more complex cognitive processes involved in solving a previously unseen 'code intent' question than for an unseen tracing question (Lister et al. 2006). Additionally they found that even experts sometimes approach 'code intent' questions by first partially tracing the code in order to discover the code's purpose. In classifying questions to Bloom the highest cognitive process level necessary to solve the problem should be used. Consequently, at the lowest possible level 'code intent' questions must be *apply*. We believe that code intent tasks are more complex than has previously been assumed. It is likely that students first break down the code into manageable chunks and then try to determine the goal of each chunk, possibly by using a tracing strategy. At this point it is likely that they try to start mapping this code to their existing knowledge. Subsequently the students try to establish how the parts relate to one another and attempt to arrive at an overall purpose for the code. If this viewpoint is accepted then it is evident that 'code intent' questions require the students to be thinking at the *analyse* level. This classification would be more in line with the SOLO and Block classifications for 'code intent' questions and would better reflect the level of difficulty of such questions for novice programmers.

3.2 Reflections on the Block model

The Block model classification of this small set of exam questions seems to indicate that there is a relationship between the Block classification of a question and the observed difficulty of a question.

The average % of fully correct answers for all questions classified into a block for each block in the Block model is shown in Figure 6. When compared with SOLO and Bloom (see Table 1) the Block model classification levels appear to more accurately match the relative difficulties of code comprehension tasks for novice programmers.

Macro structure			17%
Relations	31%	24.5%	42%
Blocks	53%	74%	60%
Atoms	59%		
	Text surface	**Execution**	**Functions**

Figure 6: Average % fully correct answers

This relationship is particularly evident when examining the results by question type. For example the tracing questions (question 7A – E) become progressively more difficult for the students to answer as the block level and knowledge dimensions increase (see Table 1 and Figure 6). However the teaching context of the knowledge required to successfully solve a question affects the difficulty of that question. The students found 7C was much easier (81% correct answers) than question 7B (64% correct). On closer examination question 7C required students to determine if a number was outside of a given range. The selection statement used a logical or. This code had been covered in detail in class using a "range doodle" (Whalley et al. 2007). Many of the scripts had such doodles on them indicating that although the code was presented as the opposite logic of the class room example, which checked if values were within a range, the teaching had an impact on the learning of the students. Question 7B on the other hand was a simple remainder operation. The fact that 36% of students could not solve this simple problem as well as they could 7C suggests that the students lack basic mathematical knowledge that was assumed in the teaching of

programming for this cohort. Despite these differences overall tracing problems which are *program execution* knowledge domain questions that were posed at the *block* level were easier that those posed at the *relations* level.

If we map the SOLO classification of our questions to the Block model classification a pattern emerges that shows a possible relationship between Block model levels and SOLO (Figure 7).

Macro structure			Relational
Relations	Multistructural	Multistructural	Multistructural
Blocks	Multistructural	Multistructural	Multistructural
Atoms	Unistructural		
	Text surface	**Execution**	**Functions**

Figure 7: Mapping of SOLO & Block model classifications

A relationship had been hypothesised by Schulte et al. (2010) and while our findings support a mapping we propose that the *relations* level actually maps to the SOLO *multistructural* level and not the *relational* (Figure 8). We found in our exam that questions at the relations level across all three knowledge dimensions were at the SOLO level of *multistructural*. It is important to note the distinction between relations (references between blocks) and 'thinking' at a *relational* level when classifying exam questions using the Block model.

Block model	**SOLO** (Schulte et al. 2008)	**SOLO** (revised mapping)
Macro	Relational	Relational
Relations	Relational	Multistructural
Block	Multistructural	Multistructural
Atom	Unistructural	Unistructural

Table 2: Mapping the Block model to SOLO

Figure 8 shows the mapping between the Bloom and Block model classifications. As observed for SOLO there is a general trend of difficulty as you progress up the Block levels and this was also reflected in decreasing student achievement.

There also appears to be a general trend of increasing cognitive complexity required to solve the questions as you move from text structure to functions across the Block model knowledge dimensions. However, this trend is not present in the student performance data on the set of questions reported in this paper. It is possible that this trend was not observed because we do not have sufficient data for some of the blocks. For some questions it was difficult to determine which block the question should be classified to if the question lay on the boundary. It may be necessary to further define the blocks and provide vignettes to guide the classification process.

Macro structure			Analyse
Relations	Understand	Apply	Analyse
Blocks	Understand	Apply	Apply
Atoms	Remember		
	Text surface	**Execution**	**Functions**

Figure 8: Mapping Bloom & Block model classifications

4 Conclusion

It is important to note that many of the limitations that exist for the use of Bloom and SOLO also exist for the Block model. In particular it is necessary to understand the context of learning and what prior exposure students have to the information required. In under taking this research we have noted that when educators attempt to design "better models" they somehow end up with models that appear to be revisions of existing taxonomies. In this case it appears that the Block model might actually be a hybrid of a revised SOLO and a revised Bloom's taxonomy.

Based on our experience SOLO still seems to the most straightforward model to apply but in using SOLO we lose the granularity to examine programming exam questions because those tasks are largely *multistructural*. A recent survey of first year programming exams found that 20% of questions in CS1 courses were tracing questions and 9% were explain questions (Simon et al. 2012). The main advantage of the Block model is that it provides us with a way of describing these novice programming tasks that gives us a level of granularity which allows us to distinguish between similar tasks in a way that SOLO or Bloom's taxonomy cannot.

The mapping of tasks to the Block model reveals 'holes' in the coverage of our examination of code comprehension. We do not have questions that are about the execution and functions of atoms or questions that require text surface and execution knowledge at the macro structure level. Examinations reflect the focus of our teaching. The lack of coverage of the Block model leads us to question whether or not we have it right. Could we be missing key tasks that might enable student learning? If we do cover the entire Block model can we improve code comprehension? Perhaps an increased focus on these missing areas during instruction will help students to develop advanced understanding more rapidly.

The work reported here is a preliminary look at the usefulness of the Block model for measuring and evaluating programming tasks and also for investigating the cognitive processes employed by students to solve the problems. In order to explore this further we intend to analyse a larger set of examination questions. We also plan to use the Block model to design assessment tasks and to attempt to establish the level at which the students are actually operating by using think-out-loud interviews.

In our analysis we have omitted code writing tasks, largely because the model was originally designed for comprehension tasks. But it would be interesting to revisit the Block model with a focus on code writing tasks. We believe that the Block model, with minor refinements, might also provide a useful framework for research and teaching of code writing tasks.

5 References

Abran, A., Moore, J., Bourque, P., DuPuis, R. and Tripp, L. (2004): Guide to the Software Engineering Body of Knowledge - 2004 Version SWEBOK ®, Los Alamitos, CA , IEEE-CS - Professional Practices Committee.

Alaoutinen, S. and Smolander, K. (2010): Student Self-Assessment in a Programming Course Using Bloom's

Revised Taxonomy. *Proc. of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*, 155–159. ACM Press.

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

Barnes, D.J. and Kolling, M. (2006): Objects First with Java: A Practical Introduction using BlueJ (3rd Edition). England, Pearson Education Ltd.

Biggs, J. B. and Collis, K. F. (1982): E*valuating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York. Academic Press.

Bloom, B. S. (1956): *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley.

Bower, M. (2008): A Taxonomy of Task Types in Computing. *SIGCSE Bulletin*, **40**(3): 281–285.

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. *Proc. of the 21st Annual NACCQ Conference*, Auckland, New Zealand, 23-30.

Denny, P., Luxton-Reilly, A. and Simon, B. (2008): Evaluating a New Exam Question: Parsons Problems. *Proc. of the 2008 International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia, 113-124.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D. Hernán-Losada,I., Jackova, J., Lahtinen, E., Lewis, T. L. McGee Thompson D., Riedesel, C. and Thompson E. (2007): *Developing a computer science-specific learning taxonomy*. SIGCSE Bull. **39**(4): 152-170.

Gluga, R., Kay, J., Lister, R., Kleitman, S. and Lever, T. (2012): Overconfidence and confusion in using Bloom for programming fundamentals assessment. *Proc. of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12)*, 147–152: ACM Press.

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

Johnson, C. G. and Fuller, U. (2006): Is Bloom's taxonomy appropriate for computer science. In A. Berglund (Ed.), *Proc. of the 6th Baltic Sea Conference on Computing Education Research (Koli Calling 2006)*, Koli National Park, Finland, 120-123.

Khairuddin, N. N. and Hashim, K. (2008): Application of Bloom's taxonomy in software engineering assessments. *Proc. of the 8th conference on Applied computer science* (ACS'08), World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 66-69.

Kintsch, W. (1998): *Comprehension: a paradigm for cognition.* Cambridge University Press.

Kropp, R. P. and Stroker, H. W. (1966): *The construction and validation of tests of the cognitive processes as described in the taxonomy of educational objectives.* Florida State University, Institute of Human Learning and Department of Educational Research and Testing.

Lahtinen, E. A. (2007): Categorization of Novice Programmers: A Cluster Analysis Study. *Proc. of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, 32-41. Joensuu, Finland.

Lister, R. (2001): Objectives and Objective Assessment in CS1. *Proc. of the thirty-second SIGCSE technical symposium on Computer Science Education (SIGCSE '01)*, 292-296: ACM Press.

Lister, R. and Leaney, J. (2003):First Year Programming: Let All the Flowers Bloom. *Proc. of the 5th Australasian Computing Education Conference (ACE2003)*, Adelaide, Australia, 221-230.

Lister, R., Simon, B., Thompson, E., Whalley, J.L. and Prasad, C. (2006): Not seeing the forest for the trees: novice programmers and the SOLO taxonomy, *Proc. of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '06)*, Bologna, Italy, 118-122.

Lopez, M., Whalley, J., Robbins, P. et al., (2008): Relationships between reading, tracing and writing skills in introductory programming. *Proc. of the 4th International Computing Education Research Workshop (ICER 2008)*. Sydney, Australia, 101-112.

Oliver, D., Dobele, T., Greber, M. and Roberts, T. (2004): This course has a Bloom Rating of 3.9. *Proc of the 6th Australasian Computing Education Conference*, Dunedin, New Zealand, 227-231,

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2010): Learning Computer Science Concepts with Scratch. *Proc. of the 6th International Computing Education Research Workshop (ICER 2010)*. Aarhus, Denmark, 69-76.

Parsons, D. and Haden, P. (2006): Parson's programming puzzles: a fun and effective learning tool for first programming courses. *Proc. of the 8th Australian conference on Computing Education, Darlinghurst, Australia*, 157–163.

Schulte, C., Busjahn, T., Clear, T., Paterson, J. and Taherkhani, A. (2010): An introduction to program comprehension for computer science educators. *Proc. of the 2010 ITiCSE Working group reports (ITiCSE-WGR'10),* Ankara, Turkey, 65-86.

Schulte, C. (2008): Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. *Proc. of the 4th International Workshop on Computing Education Research (ICER 2008)*, Sydney, Australia, 149-160.

Scott, T. (2003): Bloom's taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, **19**(1): 267-274.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E. and Whalley, J. L. (2008): Going SOLO to assess novice programmers, *Proc. of the 13th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE'08)*, Madrid, Spain, 209-213.

Shuhidan, S., Hamilton, M. and D'Souza, D. (2009): A taxonomic study of novice programming summative assessment. *Conferences in Research and Practice in Information Technology*, *95*: 147-156.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J. and Warburton, G. (2012): Introductory programming: examining the exams. *Proc. of the 14thAustralasian Computing Education Conference (ACE2012)*, Melbourne, Australia, 61-70.

Starr, C. W., Manaris, B. and Stalvey, R. H. (2008): *Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science*. SIGCSE Bulletin, **40**(1): 261–265.

Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M. and Robbins, P. (2008): Bloom's Taxonomy for CS assessment. P*roc. 10th Australasian conference on Computing Education (ACE 2008)*, Wollongong, NSW, Australia, 155-162.

Whalley, J., Clear, T. and Lister, R. (2007): *The many ways of the BRACElet project*. Bulletin of Applied Computing and Information Technology, **5**(1). Retrieved August 3, 2012 from http://www.naccq.ac.nz /bacit/0501/2007Whalley_BRACELET_Ways.htm

Whalley, J., Prasad, C. and Kumar, P. K. A. (2007): Decoding doodles: novice programmers and their annotations, P*roc. of the 9th Australasian conference on Computing Education*, Ballarat, Victoria, Australia, 171-178.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A. and Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proc. of the 8th Australasian Computing Education Conference (ACE2006)*, Hobart, Australia, 243-252.

Whalley, J., Clear, T., Robbins, P., and Thompson, E. (2011): Salient Elements in Novice Solutions to Code Writing Problems. *Conferences in Research and Practice in Information Technology*, **114**: 37-46.

Appendix

Question 4

```
import  java.util.ArrayList;
public  SimpleShop{

private  String  sName
private  String  sPhoneNumber;
private  String  aAddress;
private  ArrayList  lstInventory;
private  double  dTotalAmountSold;

public  SimpleShop(String  name,  String  address {
   aAddress  =  address;
   sName  =  name;
   lstInventory  =  new  ArrayList();
   dTotalAmountSold  =  "0.0";
}

public  String  getAddress(){
   Return  sAddress;
}

public  int  getPhoneNumber(){
   return  sPhoneNumber;
}

public int setPhoneNumber(String phoneNumber){
   sPhoneNumber = phoneNumber;
}

public void addItem(Item item){
   lstInventory.add(item);
}

public  int  numberOfItems(){
   lstInventory.size();
}

public  boolean  sell  Item(Item  item){
   boolean  bSold  =  false;
   if(lstInventory.contains(items){
      lstInventory.remove(item);
      dTotalAmountSold  +  item.getPrice();
      bSold  =  true;
   }
   return  bSold;
}

}
```

A - missing ;

A - missing)

B

A- should be return

C- wrong return type

B - should be void

B - should have return statement

A

C

A – should be +=

Question 6

Here are some lines of code that in the right order would make up a method to count the occurrences of a letter in a word.

```
if(sWord.charAt(i)  ==  c)
for(int i = 0; i < sWord.length;
i++) return count;
int  count  =  0;
public int countLetter(String sWord, char
c) count++;
```

Each box represents a placeholder for the lines of code above. Each line of code must be place in only one of the boxes.

```
{
   [                                    ]
   [                                    ]
   {
      [                                 ]
      {
         [                              ]
      }
   }
}
   [                                    ]
```

Question 7A

What are the values of a, b and c after this code is executed?

```
public void int q7A(){
   int a = 3;
   int b = 6;
   int c;
   a += 2;
   b -= 4;
   c  =  b  +  a;
}
```

Question 7B

What will this method return for each pairs of inputs shown?

```
public void int q7B(int num1, int num2){
   return num1 % num2;
}
```

num1	num2	returns
17	5	
18	6	

Give a value for each of the two input parameters that would cause the method to return the value

5: `num1....... num2.......`

Questions 7C, 7D and 7C all have the same instruction:
Complete the table below to show what this method will return for the various values shown.

Questions 7C

```
public boolean q7C(int iValue){
   boolean bValid = false;
   if(iValue>=FIRST_VAL &&
      iVALUE<SECOND_VAL){ bValid = true;
   }
   return  bValid;
}
```

iValue	FIRST VAL	SECOND VAL	returns
17	17	2	
18	17	20	
4	3	4	

Question 7D

```
public boolean q7D(int
   iLimit){ int iIndex = 0;
   int  iResult  =  0;

   while(iIndex  <=  iLimit){
      iResult  +=  iIndex;
      iIndex  ++;
   }
   return  iResult;
}
```

iLimit	returns
-1	
3	
0	

Question 7E

```
public int q7E(int[] numbers){
   int iResult = 0;
   for(int i = 0; idx < numbers.length;
      idx++){ if(numbers[idx] > iResult)
      {
         iResult  =  numbers[idx];
      }
   }

   return  iResult;
}
```

numbers	returns
{1,2,3,4,5}	
{20,-10,6,-2,0}	

Question 10A

```
public double method10A(double[]
   numbers){ double num = 0;
   for(int i = 0; i < numbers.length; id++){
        num += numbers[i];
   }

   return  num;
}
```

Question 10C

```
public double method10C(int[] numbers, int
   val){ int x = 0;
   int y = numbers.length-
   1; int z, temp;
   boolean  switch  =  false;

   while (!switch && (x <= y){
      z = (x + y)/2;
      temp  =  numbers[z];
       if(val  ==  temp){
         switch  =  true;
      }
      else if(val < temp){
         y = z -1;
      }else{
         x  =  z  +  1;
      }
   }
   return  switch;
}
```