

# How difficult are novice code writing tasks? A software metrics approach

Jacqueline Whalley and Nadia Kasto

School of Computing and Mathematic Sciences

AUT University

Auckland, New Zealand

{jwhalley,nkasto}@aut.ac.nz

## Abstract

In this paper we report on an empirical study into the use of software metrics as a way of estimating the difficulty of code writing tasks. Our results indicate that software metrics can provide useful information about the difficulties inherent in code writing in first year programming assessment. We conclude that software metrics may be a useful tool to assist in the design and selection of questions when setting an examination.

**Keywords:** software metrics, code writing, novice programmers, assessment.

## 1 Introduction

There is a plethora of literature in computing education pointing to the fact that novice programmers find programming particularly difficult and that assessing the knowledge and skills the students have gained is problematic (for example see: Robins, Rountree & Rountree 2003). Historically the pass rates for students undertaking first year courses have been relatively low. This in part might be due to some difficulties related to the assessment of these courses. Whalley et al. (2006) noted that “assessing programming fairly and consistently is a complex and challenging task, for which programming educators lack clear frameworks and tools” (p. 251). More recently, Elliott Tew (2010) suggested that “the field of computing lacks valid and reliable assessment instruments for pedagogical or research purposes” (p.xiii) and Whalley et al. (2011) noted that there is a need for “more consistent and equitable designs, an improved learning experience for the novice and an overall increase in the quality of teaching and assessment of novice programmers” (p. 45).

## 2 Background

In order to design better novice programming assessments computer science educators have attempted to apply various educational taxonomies. The most commonly adopted taxonomies to date are Bloom’s (Bloom, 1956), the revised Bloom’s (Anderson et al., 2001) and the SOLO Taxonomy (Biggs & Collis, 1982). One of the strengths of the use of educational taxonomies such as Bloom’s and SOLO for guiding the design of assessment is that they are designed to consider the level

of thinking and in the case of the revised Bloom’s taxonomy also the knowledge required in order to successfully solve a problem. However, the use and interpretation of Bloom and the revised Bloom’s taxonomy has proved to be problematic (for example see: Fuller et al. 2007, Thompson et al. 2008, and Shuhidan et al. 2009). In a recent, study Gluga et al. (2012) suggested 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 tasks and the sophistication of the cognitive processes required to solve them.

Researchers have reported that SOLO can be reliably used to classify code reading questions and the student responses to those questions as long as the classifiers have a shared understanding of the application of the taxonomy to code comprehension tasks (Clear et al. 2008, Sheard et al. 2008). An initial set of guidelines and descriptors for using SOLO to classify student code writing solutions were proposed by Lister et al. (2009). However, classifying student answers to code writing tasks using this interpretation of the SOLO levels proved difficult even with these guidelines (Lister et al. 2009, Shuhidan et al. 2009). A novel combination of SOLO and Bloom’s revised taxonomy was used by Meerbaum-Salant, Armoni and Ben-Ari (2010) to guide the design of assessments.

In a more recent study, Whalley et al. (2011) found that by combining a framework of salient elements and code quality factors they were able to more clearly define the SOLO categories. Using this approach they were able to reliably apply the principles of SOLO to determine the level of a code writing task or problem. However the programming tasks that they analysed were from various programming examinations and written using pen and paper rather than a computer.

The body of research into using SOLO for classifying questions and student responses to both comprehension and writing questions has consistently reported that the higher the SOLO level of a question the more difficult, as measured by student performance, the question was (Clear et al. 2008, Sheard et al. 2008, Whalley et al. (2011)).

Although some progress has been made towards being able to classify and estimate the difficulty of code-comprehension questions “we have no reliable measure of the difficulty of code-writing questions even at the macro level” (Simon et al. 2009). While SOLO and

Bloom maybe useful, given the nature of novice code writing questions we really need reliable measures at a higher level of detail than taxonomies such as SOLO can provide in order to be able to research the nature of these questions and their role in assessment.

## 2.1 On metrics and instructional design

Starsinic (1998) used an interpretation of the English language *Flesch-Kincaid readability measure* (Kincaid et al. 1975) to produce a script (in Perl) called Fathom that was designed to automatically measure the readability of code generated by junior programmers during time critical projects.

In a later study Börstler, Caspersen and Nordström (2007) proposed that some cognitive aspects of code reading can be expressed using common software measures and explored this idea in the context of two novice code reading tasks. Their aim was to develop a reliable means of selecting appropriate code examples to help guide novice programmers' learning and to determine between good and bad examples. They surmised that a good example must be readable and comprehensible and designed a framework based on these principles. Their framework consisted of *cyclomatic complexity* (McCabe 1976), an interpretation of the English language *Flesch Readability Ease Measure* (Flesch 1948) and *Halstead's difficulty metric* (Halstead 1977).

In subsequent work a software metrics approach to informing the design of code comprehension assessments, for novice programmers, was reported by Kasto and Whalley (2013). This work adopted a goal oriented approach to the identification of software metrics for measuring the difficulty of code comprehension tasks. In this study the difficulty of a question was represented as the percentage of fully correct answers provided. Novel dynamic metrics were designed specifically to measure the complexity of code tracing tasks and were shown to correlate, along with *cyclomatic complexity* and *average block depth*, significantly with the difficulty of the task. They also investigated the use of metrics for explain in plain English (EipE) questions but did not find any significant correlations between difficulty and *Halstead metrics* or *cyclomatic complexity* but noted that this may have been an artefact of the assessment questions that were used in the study. The authors concluded that software metrics may be a useful tool to assist in the design and selection of questions when setting an examination and that code writing tasks might also be amenable to the same approach by identifying relevant software metrics and applying them to the model answer and to the student solutions.

In this paper we report on preliminary attempts to use software metrics as a way of estimating the difficulty of code writing tasks.

## 3 Software Metrics

"Good code is short, simple, and symmetrical – the challenge is figuring out how to get there". –Sean Parent  
There are no software metrics that measure code which has yet to be written. Because we are aiming to develop an objective means of measuring the difficulty of a

novice code writing task prior to the students undertaking the task we elected to use the instructor's model answer as the code for which the metrics are calculated. While the model answer might provide a better quality solution that solution might actually have less complex code than many of the answers elicited from the students. In order to write the better answer the students may have to produce a more generalised, connected or integrated solution that reduces redundancy (Whalley et al. 2011). The challenge for developing a metric, for measuring the difficulty of a code writing task designed for novice programmers, is finding a measure that measures the level of quality of the code not just the structure of the code. This view is supported by Börstler, Caspersen and Nordström (2007) who reported that measures, for code examples, that are suitable for use in an educational context must also take into account factors such as level of thinking required and cognitive load. This must also be the case for code writing tasks.

### 3.1 Code structure metrics

When writing code it is necessary to come up with a structure for the code. Regardless of the quality of the solution we expect that some code structure metrics should have some relationship to the relative difficulty of code writing tasks.

The software metrics that have been shown to correlate to code tracing task difficulty measure the structure of the code and/or the data flow of the code when executing. These metrics are:

- McCabe's *cyclomatic complexity* (McCabe 1976),
- *average nested block depth*,

and two novel "dynamic metrics" for code tracing tasks (Kasto and Whalley, 2013):

- *Sum of all operators in the executed statements*
- *Number of commands in the executed statements.*

In tracing code only the paths of code that the students must trace though are adding to the complexity. In the case of code writing all paths are important so the dynamic metrics are not considered to be as relevant for code writing questions. As a consequence the metrics we selected for code writing were *cyclomatic complexity* and *average nested block depth*.

It is important to acknowledge the limitations of these metrics. *Cyclomatic complexity* in particular has been the subject of considerable criticism (for details see Shepperd 1988, Piwaowski 1982, and Magel 1981). *Cyclomatic complexity* "directly measures the number of linearly independent paths through a program's source code" (McCabe 1976). However in calculating *cyclomatic complexity* statements such as else, do and try, object creation and method calls are not considered. It is highly likely that these statements contribute to the complexity of a code writing task for novice programmers. However given that *cyclomatic complexity* and *average nested block depth* were found to correlate with the difficulty of code comprehension tasks we elected to evaluate them again here for code writing tasks.

Driven by the limitations of common software complexity metrics, Magel (1981) proposed a complexity metric based on regular expressions. Magel represented the structure of a piece of code as a control flow graph

and then derived a regular expression from the control flow graph. The symbols in the regular expression were then counted to give the complexity structure metric. Full details of the calculation of this metric can be found in Magel's paper. Magel surmised that "confusing program segments require longer regular expressions" and therefore a higher value for his metric (p.63). Because the quality of the model solution may prove to be more predictive of difficulty than the structure of the code we also selected Magel's *regular expression metric* for evaluation. We hypothesised that questions that provide the opportunity for solutions that are more refined (more generalised, connected or integrated) have a higher *regular expression metric* and are likely to be more difficult questions than those that do not have the potential for refinement.

Additionally we used the following structural metrics:

- The total *number of commands*; the number of java method calls.
- The total *number of operators*
- The *number of unique operators*

We included the *number of commands* because as an artefact of using a micro-world almost all of the procedures written required the students to call methods on objects. The *number of commands* metric measures the number of java methods called in the model answer. Both *number of operators* and *number of unique operators* were included because we were interested to know whether it is the total number or the number of different operators required that increases the difficulty.

### 3.2 Code readability metrics

A basic prerequisite for understandability is readability (Börstle, Caspersen and Nordström 2007). In order for code to be readable the basic syntactical elements must be easy to recognize. Only then, can relationships between the elements be established which may then lead to an understanding. It is reasonable to include a metric that measures the readability of code (i.e. of the model answer for a novice programming question) because empirical research has found that there is a strong relationship between the ability to explain code and write code with pen and paper (Lopez et al., 2008).

Readability metrics have been developed and applied to natural languages. These language measures generally produce a single numeric value, which indicates either the grade level (1-12) or readability (usually 1-100) of a document and which is constructed from the average number of syllables per word and the average number of words per sentence.

Although these natural language metrics are far from perfect, and despite their apparent simplicity, they have been found to be useful in practice. One of the most commonly used measures, the Flesch-Kincaid metric (Flesh 1948) is integrated into popular text editors and has been in used for over 50 years. However, these measures don't map well onto code therefore simply running a prose-readability test on student code would not generate a useful measure (Starsinic 1998).

The Software Readability Ease Score (SRES) is an adaptation of the Flesch Reading Ease Score where the lexemes of the programming language are interpreted as syllables, its statements as words, and its units of

abstraction as sentences (Börstler, Caspersen and Nordström 2007). This metric was designed on the premise that the smaller the average word length and the average sentence length, the easier it is to recognize relevant chunks (units of understanding). Unfortunately the authors did not provide the detail for the calculation of the metric.

Starsinic (1998) developed a similar metric where he opted to measure the number of tokens per expression (e.g. ++, ; , {, && and any keyword) , the number of expressions (e.g. 0.2 and (\$a + 6)) per statement (e.g. a = \$foo::bar \* 7;) and the number of statements per Perl subroutine. His final formula was; code complexity =

$$\begin{aligned} & ((\text{average expression length in tokens}) * 0.55) \\ & + ((\text{average statement length in expressions}) * 0.28) \\ & + ((\text{average subroutine length in statements}) * 0.08). \end{aligned}$$

The paper concluded that a low Starsinic readability metric value indicates a more readable piece of code and that a piece of code with a readability of 2.91 was very readable whereas code with a readability of 6.85 was considered to be very complex and therefore hard to read.

No justification or explanation is provided for the weightings given to each operand in the formula or for the thresholds that were used to determine the relative level of complexity of the code readability.

We elected to start from Starsinic's readability metric but we altered the way in which expressions are counted. For example, in Starsinic's method an expression such as `n=n+1;` would count as one expression but we counted this as two expressions in an attempt to more closely map the way in which a novice might read the expression. We think it is likely that a novice would break this down into two expressions firstly evaluating `n+1;` and then evaluating the assignment.

## 4 Dataset

The eleven code writing questions analysed in this study were selected from a series of controlled, summative practical programming tests held throughout the first semester of a first year Java programming course. The course adopts a back to basics procedural approach (similar to that suggested Reges (2006)) except that the learning is supported by an in-house micro-world called *Robot World* in the BlueJ IDE. For each question the students were provided starting code with unit tests, as a BlueJ project, and asked to add a method to that project (see Appendix A for the questions). Sixty student responses were analysed for each question. These students had given ethical consent for their data to be used and were representative of the entire cohort.

## 5 Analysis

Table 1 gives the software metrics and student performance for each of the questions analysed. It should be noted in interpreting the analysis that difficulty is being measured as the percentage of fully correct answers. For example question 11 is the easiest question with a percentage difficulty of 100% whereas question 1 was the most difficult question with 14% of students giving a correct working solution.

	Questions										
	1	2	3	4	5	6	7	8	9	10	11
Difficulty (%) (n = 60)	14	24	39	52	55	63	84	84	90	98	100
<i>cyclomatic complexity</i>	12	5	5	5	6	5	4	3	2	2	1
<i>average nested block depth</i>	4	2	3	2	4	2	2	2	2	2	1
<i>number of operators</i>	18	15	4	14	8	8	3	6	1	1	0
<i>number of unique operators</i>	5	8	2	6	4	4	2	6	1	1	0
<i>number of commands</i>	49	13	14	27	20	20	9	7	3	4	4
<i>regular expression metric</i>	60	24	24	29	31	25	20	14	8	8	3
<i>readability metric</i>	5.78	4.88	2.74	1.78	2.38	4.20	1.69	1.90	1.14	1.33	1.28

Table 1: Metrics for the instructor's model answer for each question

*Cyclomatic complexity*, *average nested block depth*, *number of operators* and *number of unique operators* were calculated using the standard procedures provided by the Rationale® Software Analyzer 7.1 (RSA 2013) tool. The *regular expression metric* and the *readability metric* were calculated by hand.

The significance of the correlation of each metric to the difficulty of each question was then tested using a Pearson's correlation (Table 2).

*Cyclomatic complexity*, the *regular expression metric* and the *readability metric* were found to all correlate strongly with the difficulty of the novice code writing questions that we analysed in this study.

The higher the *cyclomatic complexity*, the more complex the control flow of the program code is and the more difficult the question is (as evidenced by a low percentage of students getting the answer correct).

The more deeply nested the branches of the code are the higher the average nested block depth is and the more difficult the question was for the students. This is not really surprising. Research investigating student responses to code writing questions found that students find questions that can be solved by writing the code line by line with limited reference to the previous lines of code are easier than those that require the students to understand the relationship between the chunks or blocks of code that they have written (Whalley et al. 2011).

software metric	Pearson's correlation	
	<i>r</i>	<i>p</i>
<i>cyclomatic complexity</i>	-0.848	0.0009
<i>average nested block depth</i>	-0.647	0.0313
<i>number of operators</i>	-0.836	0.0013
<i>number of unique operators</i>	-0.644	0.0321
<i>number of commands</i>	-0.763	0.0062
<i>regular expression metric</i>	-0.839	0.0012
<i>readability metric</i>	-0.906	0.0001

Table 2: The correlations between metrics and difficulty

The number of operators, in the case of the questions analysed here, correlates more strongly with difficulty than the number of unique operators. The opposite was found for code tracing questions where the unique operators correlated more strongly to difficulty (Kasto and Whalley, 2013). The repetition of operators perhaps doesn't contribute to the complexity of the task but it does correlate to the student difficulty measure so perhaps it gives them more opportunity to make mistakes.

Similarly we found that the higher the number of Java commands required the more difficult the question is.

For the regular expression metric a higher value results from nested code (Figure 1, A vs. B), backward branches rather than forward branches (Figure 1, D vs. C) and increasing complexity in selection statements (Figure 1, E and F).

The strong correlation between difficulty of the question and increasing structural and data flow complexity, as measured by the regular expression metric, confirms our original hypothesis and supports the conjecture that many students cannot write code that requires more complex structures and that there must be some relationship between the ability to design code structure and being able to produce working code regardless of the quality of their code.

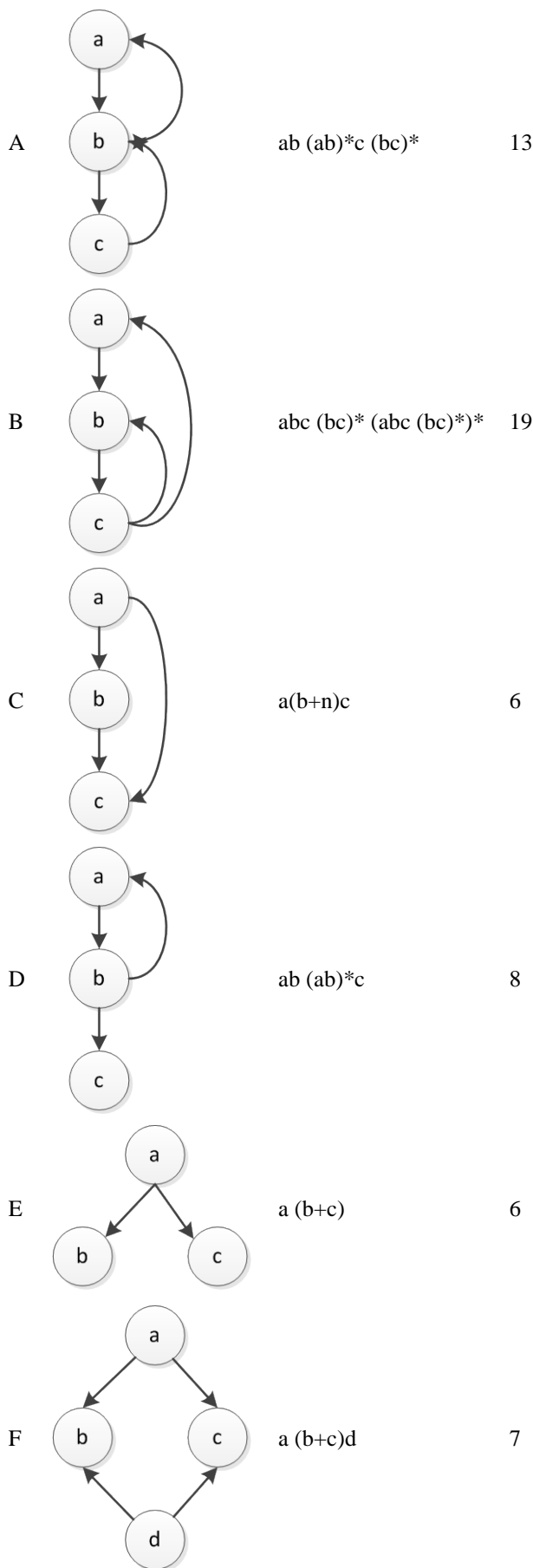
Given that we are analysing the instructor's model answer, we are assuming that it is good code. If there are, for example, nested blocks to reach this solution a relatively high level of integration of the code and merging of plans is likely to be required. For such a question there are usually several solutions that could provide a working solution. If the student's solutions are of a lower quality than the instructor's code then you could argue that the code produced by them is more confusing and that the students would find it hard to correct any errors in their code. This could make the question more difficult for the novice programmer than the analysis of the model answer would indicate.

The readability measure also correlates strongly to difficulty. The easier the model answer code is to read the easier the code is to write. It is possible that there is a causal relationship between readability of code and the ease of writing.

## 6 Limitations

While the findings of this study are encouraging there are some caveats.

We have only examined a relatively small set of code writing questions. The questions were selected to cover the key topics taught in our first year programming course; sequence, selection and iteration. The sequencing of the questions within the tests could add to our observed difficulty of the question. However with such strong correlations it is unlikely that this effect would significantly alter our findings.



**Figure 1: Flow graphs, regular expressions and regular expression metric exemplars**

The questions we have analysed are also limited to “unseen” questions presented to student in a test situation. If previously seen questions are included it is likely correlations with the metrics used here will be less significant or even not significant. The difficulty of the question would be affected by the level of thinking required. A problem for which the students have already seen the code may mean that students can simply answer the question by recall.

Much of the reasoning around why we are seeing the relationships between the metrics and actual difficulty is based on conjecture and this aspect of the work could be improved by observing the students in the tests.

Some of the metrics used in this study may not be generalizable to all teaching contexts or indeed to all novice programming tasks. Courses that adopt an objects first pedagogy may have writing tasks for which other object orientated metrics might be applicable such as cohesion and coupling metrics. For a back to basics, algorithm focused, java course that does not utilise micro worlds but instead uses a typical IDE metrics such as number of commands may not be relevant. It is worth noting that for most metrics the range of values in a typical novice code writing task is likely to be relatively small. For example *average nested block depth* where deep nesting may be discouraged, by the instructor, in favour of separation of inner blocks into method calls. Despite the relatively small range of values we have found the metrics still correlate strongly with difficulty.

In selecting the metrics to use we believe that average nested block depth, cyclomatic complexity, regular expression metric and readability should provide a measure of difficulty of the task regardless of teaching approach and programming context. However other metrics would need to be selected based on the teaching approach. Some aspects of the teaching approach will be reflected in the model answer. For example, if considering a typical programming task such as printing a box of asterisks of any size the model answer may be a solution that has two for loops while another instructor’s model answer may consist of a nested loop.

While you could argue that as experienced teachers we consider these aspects of a programming problem when setting an assessment it is still useful to have a method for objective evaluation of the difficulty of a code writing question prior to including it in an assessment.

## 7 Future work

Where to from here?

Further analysis could be undertaken to examine which metrics are general predictors of difficulty of novice programming tasks. Moreover metrics could be identified that are useful for specific pedagogies.

If we can establish suitable heuristics for selection of metrics for a given course it may be possible to use this approach to automatically grade code writing tasks. We may even be able to use metrics as a tool for providing immediate feedback to the students about the quality of their solutions. Good code must be simple, readable and comprehensible and we want our students to be producing quality code. However in this study, we do not consider the quality of the students solutions in determining the difficulty of a question – a fully correct

answer may not be a well-designed answer. If you were wishing to adopt metrics to assist in the grading of student work then perhaps some measure of distance of the student's answer from the instructor's model answer might be useful. Some work has been undertaken which investigates the usefulness of software metrics as a form of formative feedback for novice programmers (Cardell-Oliver 2011). This work used program size metrics, unit tests and program style violation counts as forms of automated feedback. While software metrics such as the ones we have explored in this paper are difficult for novice programmers to interpret directly, if supplied with guidelines for interpretation it is possible that students might also find them a form of useful feedback.

Finally we believe that this approach has value as a research tool and provides a way of comparing questions in an empirical manner. However, we concur with Börstler, Caspersen and Nordström (2007) that measures that are suitable for use in an educational context must also take into account factors such as level of thinking required, cognitive load and instructional design.

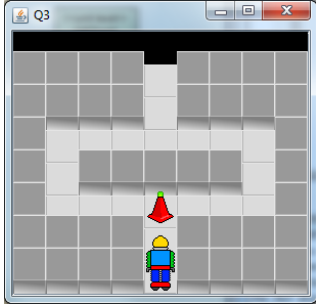
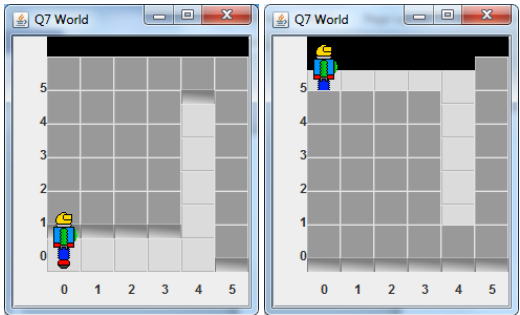
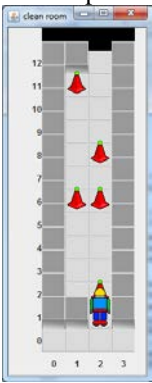
Metrics should not be used as a silver bullet but used in conjunction with more subjective measures of difficulty such as SOLO or Bloom's classification which consider the level of thinking and/or knowledge required.

## 8 References

- Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., et al. (2001): *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman.
- Biggs, J. B. and Collis, K. F. (1982): *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York. Academic Press.
- Bloom, B. S. (1956): *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley.
- Börstler, J., Caspersen, M.E. and Nordström, M. (2007): Beauty and the Beast — Toward a Measurement Framework for Example Program Quality, Technical Report, Department of Computing Science, Umeå University, ISSN 0348-0542.
- Cardell-Oliver, R. (2011): How can Software Metrics Help Novice Programmers? *Proc. of the 13th Australasian Computing Education Conference (ACE 2011)*, Perth, Australia. 55-62.
- Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., et al. (2008): Reliably Classifying Novice Programmer Exam Responses using the SOLO Taxonomy. *Proc. 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008)*, Auckland, New Zealand, 23--30.
- Gluga, R, J Kay, R Lister, S Kleitman, and T Lever (2012): Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals. *Proc. 14th Australasian Computing Education Conference (ACE 2012)*, Melbourne, Australia, 147-156.
- Elliott Tew, A. (2010): Assessing fundamental introductory computing concept knowledge in a language independent manner. PhD dissertation, Georgia Institute of Technology, USA.
- Flesch, R. (1948): A new readability yardstick. *Journal of Applied Psychology*, **32**: 221-233.
- Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., et al. (2007): *Developing a Computer Science-specific Learning Taxonomy*. *ACM SIGCSE Bulletin*, **39** (4):152-170.
- Halstead, M.H. (1977): *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA, Elsevier Science Inc.
- Kasto, N and Whalley J. (2013): Measuring the difficulty of code comprehension tasks using software metrics. *Proc of the 15th Australasian Computer Education Conference (ACE 2013)*, Adelaide, Australia, 57-6.
- Kincaid, J. P., Fishburne R. P. Jr., Rogers R. L. and Chissom B. S. (1975): Derivation of new readability formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy enlisted personnel. Research Branch Report 8-75, Millington, TN: *Naval Technical Training, U. S. Naval Air Station, Memphis, TN*. <http://digitalcollections.lib.ucf.edu/u/?IST,26253>. Accessed 15 August 2013.
- Lister, R., Clear, T., Simon, B. Bouvier, D.J., l Carter, P. et al. (2009): Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bulletin*, **41**(4): 156-173.
- Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2010): Learning Computer Science Concepts with Scratch. *Proc. of the 6th international workshop on Computing Education Research (ICER10)*, Aarhus, Denmark, 69-76.
- McCabe T. (1976): A Software Complexity Measure, *IEEE Transactions on Software Engineering*, **SE-2**(4): 308-320.
- Magel, K. (1981): Regular expressions in a program complexity metric. *Sigplan Notices - SIGPLAN*. **16**(7): 61-65.
- Piowowski, P. (1982); A nesting level complexity measure. *Sigplan Notices - SIGPLAN*, **17**(9): 44-50.
- Robins, A., Rountree, J. and Rountree, N. (2003): Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, **13**(2): 137-172.
- RSA, IBM. [http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp?topic=/com.ibm.iea.rsar/plu gin\\_types.html](http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp?topic=/com.ibm.iea.rsar/plu gin_types.html). Last accessed 8 August 2013.
- 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.
- Shepperd, M., (1988): A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, **3** (2): 30-36.

- 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, B., Lopez, M., Sutton, K., and Clear, T. (2009): Surely we must learn to read before we learn to write!. *Conferences in Research and Practice in Information Technology*, **95**: 165-170.
- Starsinic, K. (1998): Perl Style. *The Perl Journal*, Fall 1998. **3**(3), [http://www.foo.be/docs/tpj/issues/vol3\\_3/tpj0303-0006.html](http://www.foo.be/docs/tpj/issues/vol3_3/tpj0303-0006.html)
- Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M., and Robbins, P. (2008): Bloom's Taxonomy for CS assessment. *Conferences in Research and Practice in Information Technology*, **78**: 155-162.
- Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P. and Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Australian Computer Science Communications*, **52**: 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 A: The questions

	appropriate exception should be thrown. If the robot encounters a wall before moving the full distance it should stop rather than crashing. The method should return true only if the robot moved the full distance.
3	<p>In this question the students must write code to move the robot from a set starting location at (4, 0) to a fixed exit at location (4, 6). In order to do this the robot must choose one of two paths. If there is a beeper at the first intersection (4, 2) then the robot must follow the eastern path otherwise it should follow the western path.</p> 
4	In this scenario there are two corridors with a gap between them. The length of each of the corridors changes randomly every time the World is created, but the gap is always in the same location.
5	The students were provided with a method header and asked to write a summing algorithm; write code that makes a robot move forwards until it reaches a wall while picking up any beepers that it encounters and then print out the total number of beepers the robot collected.
6	Complete the method findBeeper that moves the robot through a spiral maze until it reaches a beeper. You should also count how many steps the Robot navigate to the beepers and return the number of steps required.
7	<p>A robot starts in one of two possible initial states, as shown in the figures below:</p>  <p>Write a program to move the robot to the end of the corridor. If the robot starts at location (0, 0), it must finish at location (4, 4) facing north. If the robot starts at location (0, 5), it must finish at location (4, 1) facing south.</p>
1	<p>This question asks the students to write a method that makes the robot clean the room. <i>The robot must pick up all the beepers left lying around and if there are enough beepers to fully load the beeper washer (at location (2, 12) ) any remaining beepers should be neatly placed at location (2,0).</i> The students are supplied with the method signature and unit tests to test that the beepers have been dropped at the appropriate location(s). The tests include starting worlds with 0, 5, 9, 10, 15 and 20 beepers.</p> 
2	This question asks the students to write a method called advanceRobot that has two parameters a Robot and a distance to travel (the number of cells that the robot should advance). <i>The robot should only be able to move if it is alive and if the distance to travel is positive if it is unable to move an</i>

8	In this question the students are provided with a robot in a cell that contains a number of beepers. The students are asked to write a method called <i>pickUpNBeepersCheckIfAll()</i> that takes an integer parameter, and makes the most recently created robot pick up that number of beepers from the beeper stack at its current location. You can assume that there are enough beepers in the stack for the robot to do this safely. The method should return true if the robot has picked up all the beepers at its current location, or false if there are still beepers on the ground.
9	Write a method called <i>pickUpBeeperStack()</i> that makes the most recently created robot pick up all the beepers at its current location. The method should return no value and take no parameters.
10	For this question the students are supplied with the method header they are asked to complete the method body so that <i>the robot turns left then if there is no wall in the way moves forward one cell.</i>
11	For this question the students are supplied with the method header they are asked to complete the method body by writing <i>a sequence of three statements to make the robot drop the beeper it is carrying, then move the robot forward one cell and turn the robot left once.</i>