Measuring the difficulty of code comprehension tasks using software metrics

Nadia Kasto and Jacqueline Whalley
Software Engineering Research Laboratory
School of Computing and Mathematical Sciences
AUT University
PO Box 92006, Auckland 1142, New Zealand
{nkasto,jwhalley}@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 comprehension tasks. Our results indicate that software metrics can provide useful information about the difficulties inherent in code tracing 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 comprehension, novice programmers, assessment.

1 Introduction

It is common knowledge that novice programmers find programming particularly difficult and that assessing the knowledge and skills the students have gained is problematic. 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).

In order to write better questions and assessments computer science educators have attempted to apply various educational taxonomies to guide the design of assessments. In 2006 an analysis of a program comprehension question set within two key pedagogical frameworks: the Bloom (Anderson et al. 2001) and SOLO (Biggs and Collis 1982) taxonomies was reported (Whalley et al. 2006). It was found that student performance was consistent with the cognitive difficulty levels, indicated by the assigned Bloom category of the questions. Additionally a degree of consistency was found between student performance and the SOLO taxonomy level of their responses to an 'Explain in Plain English' (EiPE) question. While these results and results of subsequent studies by the Bracelet project team were encouraging (e.g.: Lister et al. 2006, Thompson et al. 2008, Clear et al. 2008, Sheard et al. 2008, Whalley et al. 2011) many educators have reported difficulties in reliably using these and other taxonomies in the context of novice computer programming assessment design, evaluation and research (e.g.: Fuller et al. 2007, Thompson et al. 2008, Shuhidan, Hamilton and D'Souza 2009, Meerbaum-Salant, Armoni and Ben-Ari 2010)

An alternative or supplementary approach to informing the assessment instrument design process might be to use software metrics in order to determine the difficulty of examination questions designed to assess novice programmers.

2 Background

Typically research into software metrics is conducted in the context of relatively large scale commercial software development projects. However some work using software metrics to support research related to the improvement of teaching and learning of computer programming has been undertaken.

One study applied software metrics to previously reported code used in empirical studies of novice and expert program comprehension (Mathias et al. 1999). The metrics were used in order to examine the underlying nature of code designed to study the process of program comprehension. The software metrics used in this study were *lines of code* and *cyclomatic complexity* (McCabe 1976) . A correlation was found between the complexity of the code and the comprehension strategies observed by the original researchers suggesting that *lines of code* and *cyclomatic complexity* might correlate to the difficulty of small program comprehension tasks.

Parker and Becker (2003) employed Halstead's metrics (Halstead 1977) to measure and compare the effectiveness of students solutions of two different code writing assessments based on the premise that the metrics can be seen as a measure of work done. An earlier empirical study measuring student solutions to code writing questions using software metrics and comparing those measures with student performance found that neither *lines of code* nor Halstead's metrics were able to predict the error rate in the student's solutions (Klemola 1998). Subsequently, Klemola and Rilling (2003) developed a software metric called the *Kind of Line of Code Identifier Density* (KLCID) metric for analysing the cognitive complexity of program comprehension tasks. KLCID was designed to capture the effect of the number

of unique kinds of code lines in a program segment. For KLCID only conceptually unique lines of code are counted and within these unique lines the identifier density is calculated (Klemola and Rilling 2003). The effectiveness of the KLCID metric was evaluated in a study of code comprehension tasks from a final examination of an introductory C++ course. The complexity of each task as measured by KLCID was compared with the average student performance on the task. A correlation was found between increasing KLCID and decreasing student performance. This finding is not surprising as in text comprehension it has been found that a higher density of concepts decreases the rate of comprehension (Kintsch and van Dijk 1978). The authors concluded that KLCID was "a good candidate to measure the complexity of code comprehension assessment tasks within the same course" (Klemola and Riling 2003). However the code comprehension examination questions themselves are not reported so it is difficult to determine the general applicability of the KLCID metric to novice programmer code comprehension tasks.

3 Software Metrics

In order to attempt to measure the difficulty of typical code comprehension and code tracing examination questions we first selected an appropriate set of software metrics. Software metrics focus on a particular feature of a program and are often devised with a single programming paradigm in mind. Table 1 shows a set of commonly employed software metrics classified by metric type and their applicability to three programming paradigms.

The examination questions that we have analysed are from a CS1 (first semester) Java programming course. The questions are typical code tracing and EiPE questions that have been reported extensively in the recent literature (e.g.: Venables, Tan and Lister 2009, Murphy, McCauley and Fitzgerald 2012).

Metric Type	Metric	Programming Paradigm		
		imperative	structural	object oriented
Basic	Number of lines of code	✓	✓	✓
	Number of *blank* lines of code	✓	✓	✓
	Number of comment lines of code.	✓	✓	✓
	Number of comment words.	✓	✓	✓
	Number of statements	✓	✓	✓
	Number of methods.		✓	✓
	Average line of code per method.		✓	✓
	Number of parameters.	✓	✓	✓
	Number of import statements.		✓	✓
	Number of arguments.		✓	✓
	Number of methods per class.			✓
	Number of classes referenced.			✓
	Average number of attributes per class			✓
	Number of constructors.			✓
	Average number of constructors per class.			✓
	KLCID	✓	✓	✓
Complexity metrics	Cyclomatic complexity	✓	✓	✓
	Nested block depth.	✓	✓	✓
Halstead metrics	Number of operands.	✓	✓	✓
	Number of operators.	✓	✓	✓
	Number of unique operands.	✓	✓	✓
	Number of unique operators.	✓	✓	✓
	Effort to implement.		✓	✓
	Time to implement.		✓	✓
	Program length.		✓	✓
	Program level.		✓	✓
	Program volume.		✓	✓
	Maintainability index.		✓	✓
Object oriented	Weight method per class.			✓
	Response for class.			✓
	Lack of cohesion of methods.			✓
	Coupling between object classes.			✓
	Depth of inheritance tree.			✓
	Number of children.			✓

Table 1: Static metrics and their applicability across programming paradigms

Although the course is taught with an objects first approach most of the comprehension questions are small bite size pieces of code and are largely procedural. Therefore, even if the code is encapsulated in a method, many of the questions are essentially procedural in nature.

Of the metrics in Table 1 we selected the subset which we deemed to be most applicable to measuring the difficulty of novice code tracing and EiPE tasks:

- Number of statements
- Number of operands (including all identifiers that are not key words)
- Cyclomatic complexity
- Average nested block depth
- Average number of parameters

One EiPE question involved code that contained two methods and internal method calls. The object oriented metric, the *number of methods*, that had a variation in value was therefore included as part of our metric set for EiPE questions.

We did not use KCLID because most of our code comprehension questions did not contain lines of code which were not conceptually unique lines of code. Additionally, we elected not to use *the number of operators* metric as the *number of operators* is proportional to the *number of operands* and would therefore not contribute anything new to the evaluation.

We also supplemented this set of metrics with two simplified versions of dynamic metrics for the measurement of the difficulty of the code tracing questions that we have called the *sum of all operands in the executed statements* and the *number of executed program statements*. The *sum of all operands in the executed statements* was calculated as the sum of all operands (*O*) in the executed statements *ES* where the total number of executed statements is $\mathbf{v}$.

$$\textit{Sum of all operands in the executed statements} = \sum_{i=1}^{\mathbf{v}} ES_i\,(O)$$

The *number of executed program statements* was counted as the total number of statements executed for the complete tracing task. This count, if a selection or iterative statement is included in the code, is dependent on the data provided as the input for the specific tracing task.

These dynamic metrics provide a measurement of the execution complexity of the code. It seems reasonable to include such metrics because when students are tracing code they are hand executing the code and, from an initial input, processing data through the code line by line via the relevant paths of the code in order to determine the output. We postulate that these metrics will correlate well with the difficulty of the tracing task.

4 Data Sets

The questions analysed in this study were selected from several occurrences of a final examination for a first year Java programming course. The teaching team and pedagogy was the same for all instances of the course and the results were taken from exam scripts for which the students had given ethical consent for their data to be used. These students were representative of the entire cohort.

For the code tracing questions two examinations were analysed. One examination contained the questions 1A-D and resulted in 93 student responses for analysis and the other contained questions 2A-2E for which 79 student responses were analysed (Table 2). The EiPE questions were selected from three examinations. For 3A-D, 4A-C and 5A-E there were respectively 93, 79, and 92 student responses analysed. The percentage of fully correct answers is used as the measure of question difficulty.

The distribution of the percentage of fully correct answers was irregular and clustered. We therefore used natural, data driven, clustering to place the data into a five point scale from very easy (a relatively high percentage of students got the question correct) to very hard. Questions of similar difficulty, as determined by student achievement, for example 1D (26%), 2D (21%) and 2E (27%) were therefore ranked at the same difficulty level. These ranks were then used to determine whether or not there was a correlation between difficulty and the relevant metrics. It seemed unlikely that one common set of software metrics would provide useful information about different types of questions or about questions designed to measure significantly different types of knowledge. For this reason the data from the code tracing and EiPE questions were placed in separate data sets.

	questions								
	1A	**1B**	**1C**	**1D**	**2A**	**2B**	**2C**	**2D**	**2E**
Difficulty ranked	1	4	4	8	4	6	2	8	8
Cyclomatic complexity	1	2	2	3	1	1	3	2	3
Average nested block depth	1	2	2	3	1	1	2	2	3
Number of operands	14	10	12	29	13	5	13	12	17
Number of parameters	0	2	1	2	0	2	3	1	1
Number of statements	7	5	5	8	3	1	2	4	3
Sum of all operands in the executed statements	14	18	33	48	13	10	35	42	138
Number of commands in the executed statements	7	9	13	52	6	4	13	20	34

Table 2: Metrics for code tracing questions

An item discrimination analysis was undertaken to examine the relationship between student scores for each question and the total score for the related set of questions to identify any outlier questions that did not therefore belong in the data set. A point bi-serial correlation was calculated between each question and the total score, excluding the score for the question itself, for all questions in that set (tracing questions or EiPE questions) . This provides an estimate of the extent to which an individual question is measuring the same competencies as the rest of the questions in that question set. Each question is expected to contribute to the total score for that question set. Any question that does not correlate positively with the total score is probably measuring something other than what the examiners intended and does not belong in that set.

For all questions except 2C and 2B a significant positive correlation was found between students' scores on the question and their overall scores on the related set of questions. Therefore, except for 2C (r_{pb} = 0.165, p = 0.15) and 2B (r_{pb}= 0.217, p = 0.06) the questions in each set are contributing towards the respective total scores and can be considered to belong within the sets. However, the discrimination analysis also provides evidence that for some reason 2C and 2B are not measuring the same thing as the other code tracing questions. Therefore, for the purpose of further analysis, we removed both of these outliers from the data set.

The students found 2C relatively easy while we would have expected that this would be one of the more difficult code tracing questions. The students had been introduced to this code in lectures and had been guided through a similar tracing exercise with slightly different input data. Perhaps this is encouraging; clearly teaching has had some impact on student learning. Nevertheless, if test questions are set that are too close to specific examples taught in lectures they may be measuring the students' abilities to remember specific examples rather than measuring their code tracing abilities. That is, they may well be measuring something other than what the examiner intended.

On the other hand, we would have expected question 2B to be an easy question but student performance showed that they found it to be relatively difficult. Question 2B is a simple method that calculates the remainder. We believe that the issue in this question may lie with a lack of mathematical knowledge rather than a lack of programming comprehension. This conjecture is supported by the fact that many of the same students were able to answer code tracing questions that consisted of more complex code successfully. Once again it seems that the question is not measuring what the developers intended and does not belong in the data set.

5 Results

The code provided in the examination was analysed using our set of software metrics. In the case of the dynamic metrics for the code tracing questions the metrics are calculated from those parts of the code that are executed in order to arrive at the correct answer. We then compare the metrics with the student performance on the questions. The following metrics were calculated using the Rationale® Software Analyzer 7.1 (RSA 2012) tool: *number of operands*, *cyclomatic complexity*, average *nested block depth*, *average number of parameters*, and *number of methods*. Initially we calculated *lines of code*, using Rationale® Software Analyzer, as the total number of executable lines of code. In the programming examination questions the code is formatted so that the opening and closing braces are placed on their own line. Given the small size of the code for each question, lines containing only braces contribute significantly to the *lines of code* metric when calculated this way. We believe that these lines do not contribute to the complexity or difficulty of the code comprehension tasks. Consequently, we calculated the *number of statements* rather than the total lines of code.

The significance of the correlation of each metric to the categorised difficulty (encoded numerically where the easiest is ranked as 1) of each question was then tested using Kendall's **τ**-b. Kendall's **τ**-b was chosen because the datasets contained tied ranks. Table 4 gives the Kendall's **τ**-b for all the, tracing and EiPE, questions analysed.

It is worth noting that the tracing and EiPE exam questions used in this study are characterised by a low number of number of program commands and are generally confined to one or two methods. As a consequence the *cyclomatic complexity* for the exam questions does not exceed 5 and the *nested block depth* does not exceed 3.

	questions											
	3A	**3B**	**3C**	**3D**	**4A**	**4B**	**4C**	**5A**	**5B**	**5C**	**5D**	**5E**
Difficulty ranked	8	10	8	3	5.5	11.5	11.5	1	5.5	3	8	3
cyclomatic complexity	1	2	3	4.5	2	3	5	2	3	2	3	1
Average nested block depth	1	2	3	2.5	2	3	3	2	3	2	3	1
Number of operands	11	11	18	37	14	21	36	11	21	18	39	6
Number of parameters	0	2	2	2	1	1	2	2	1	2	1	0
Number of statements	5	5	6	11	5	5	16	5	5	5	9	3
Number of methods	1	1	1	2	1	1	1	1	1	1	1	1

Table 3: Metrics for 'Explain in plain English' (EiPE) questions

For code tracing questions *cyclomatic complexity*, *nested block depth* and the two dynamic metrics, developed for this study, are significantly correlated to the student performance and therefore to the observed difficulty of the question (Table 4). Increasing complexity, as defined by increasing values in the four metrics of a tracing question, therefore directly correlates with an increase in difficulty for previously 'unseen' code that does not extend beyond the courses content. The definition of 'unseen' code is code that is either entirely new code for which the key syntax and language constructs had been taught during the course or a variation on code that had been seen in the context of the course. For example the students may have, as a lab exercise, been asked to write a method that found the highest number in an array of numbers and the 'unseen' code might find the lowest number. Therefore it can be argued that the students should have the knowledge required to answer an 'unseen' question and that such a question requires them to apply or adapt their existing knowledge in order to solve the question.

Question Type	software metric	Kendall's τ-b (2-tailed)
Tracing	Cyclomatic complexity	0.775*
	Average nested block depth	0.775*
	Number of operands	0.231
	Number of parameters	0.452
	Number of java commands	0.304
	Sum of all operands in the executed statements	0.732*
	Number of commands in the executed statements	0.732*
EiPE	Cyclomatic complexity	0.289
	Average nested block depth	0.109
	Number of operands	0.219
	Number of parameters	-0.040
	Number of commands	0.274
	Number of methods	-0.277

Table 4: Correlations between software metrics and question difficulty [* $p < 0.05$]

None of the metrics used correlated significantly with the difficulty of the EiPE questions. Although it is possible that questions that require EiPE responses are inherently unsuitable for a metrics approach to predicting difficulty it is just as likely that we have yet to identify metrics capable of performing this task.

6 Conclusion

This research has analysed student responses to two types of exam questions, which are typically used in novice programming exams, code tracing and EiPE. The results have shown that some software metrics, for our dataset, correlate to the difficulty of code tracing exam questions. As a result of this study we suggest that software metrics might be a useful tool in the early prediction of the difficulty of this type of first year computer programming examination question.

More research is needed into the possible use of software metrics for evaluating EiPE questions and other forms of programming tasks and questions to see whether or not it is possible to develop metrics that are meaningful in those contexts. Further consideration needs to be given to what other metrics may be useful for the analysis of EiPE questions and perhaps to determining the criteria that should be used to determine whether or not any given EiPE question should be included in a set of questions of that type. It is possible that some of the existing metrics could provide useful information if the question set was more homogeneous.

When undertaking this analysis we found aspects of some questions that were not measured by the metrics but that affected the validity of those questions. What the question is assessing may not be what the examiner intended. For example a question that includes mathematical operators or concepts may be testing mathematical knowledge not programming knowledge. Perhaps such questions should be avoided unless the intent is to assess the mathematical concept. Additionally the use of previously 'seen' code has the potential to alter the way in which students respond to the question. An EiPE question with relatively complex code may actually be reduced to a simple recall question rather than one that requires an understanding of the code.

In this study we undertook an item discrimination analysis but it appears that some of our questions may have additional issues of validity or of inappropriate item difficulty. It is our recommendation that any future research should include a full item analysis of all questions and include only those questions that have performed adequately in terms of reliability, validity, difficulty and item discrimination in any further analysis. This would reduce the likelihood that any question set contained poorly performing questions that could obscure possible relationships between the data set and software metrics. It could also lead to the development of criteria for each question type that could be used in future to help to ensure that questions meet an appropriate standard and can be meaningfully evaluated using the appropriate software metrics.

Future work will involve applying metrics to other types of questions. This work will include measuring the contribution of each metric to the overall question difficulty with the intention of designing a single weighted metric for each question type. We also intend to verify the findings of this preliminary study firstly with a larger set of examination questions and secondly by designing questions using software metrics as a factor that is considered in that design and evaluating the effectiveness of this approach. Finally, we believe 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.

7 References

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.

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.

Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B., and Thompson, E. (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.

Elliott Tew, A. (2010): Assessing fundamental introductory computing concept knowledge in a language independent manner. PhD dissertation, Georgia Institute of Technology, USA.

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.

Halstead, M.H. (1977): *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA, Elsevier Science Inc..

Kintsch, W. and van Dijk, T.A. (1978): Towards a model of text comprehension and production. *Psychological Review*, **85**, 363-394.

Klemola, T. (1978): Software comprehension: theory and metrics. Masters Thesis, Concordia University, Montreal, Canada.

Klemola, T. and Riling, J. (2003): A cognitive complexity metric based on category learning. *Proc. of the 2nd IEEE International Conference on Cognitive Informatics (ICCI'03)*, London, UK, 106 – 112.

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

Murphy, L., McCauley, R. and Fitzgerald, S. (2012): 'Explain in plain English' questions: implications for teaching. *Proc. of the 43rd ACM technical symposium on Computer Science Education* (SIGCSE '12), 385-390

Mathias, K.S., Cross, J.H., Hendrix, T.D., and Barowski, L.A. (1999): The role of software measures and metrics in studies of program comprehension. *Proc. of the 37th Annual Southeast Regional Conference (CD-ROM)*, ACM-SE, **37**, article 13, doi =10.1145/306363.306381

Meerbaum-Salant, O., Armoni, M., and 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.

McCabe, T.J. (1976): A Complexity Measure, S*oftware Engineering, IEEE Transactions on*, **2**(4), 308- 320.

Parker, J. R. and Becker, K. (2003): Measuring effectiveness of constructivist and behaviourist assignments in CS102. *Proc. of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)*, Thessaloniki, Greece, 40-44.

RSA, IBM. http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp?topic=/com.ibm.iea.rsar/plugin_types.html. Last accessed 24 August 2012.

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.

Venables, A., Tan, G. and Lister, R. (2009): A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *Proc. of the 5th International Computing Education Research Workshop (ICER 2009)*, Berkeley, CA, USA, 117-128. Berkeley, California, August 10-11, 2009. pp. 117-128.

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**: pp. 37-46.

Appendix

Example of a typical EiPE question and a typical code tracing question:

Question 5A

In plain English, explain the purpose of this method. Note that more marks will be gained by correctly explaining the purpose of the code than by giving a description of what each line does.

```
public int method(int x, int y)
{
  int result =x;
  if(x < y)
     {
       result = y;
     }
     return result;
   }
```

Question 1C

Complete the trace table below to show what happens when this method is executed with the parameter *limit* equal to 4.

```
public int method(int limit)
{
  int result = 0;
  for(int i = 0; i<= limit; i++)
     {
       result += 2;
     }

        return result;

     }
```

i	result	
0	0	Initialisation