Using Software Metrics to Predict the Difficulty of Code Writing Questions

Said Elnaffar
Department of Computer Science and Engineering
American University of Ras al Khaimah (AURAK)
Ras al Khaimah, UAE
said.elnaffar@aurak.ac.ae

***Abstract*— Asking IT students, job interviewees, and competition contestants to write code is very common. Nevertheless, to properly assess the programming skills of such people, the anticipated difficulty level of these coding questions should be estimated and kept in mind throughout the preparation of such exams. Poor results coming out of these assessment tools may not be entirely attributed to exam takers, but rather to the poor design of the exams that fail to gauge the competency levels of each student via the ranked levels of difficulty of questions on the exam. In this research, we argue that we can develop a predictive tool, named the Predicted Difficulty Index (PDI), that is derived from the structure of the sample solutions to rank the questions based on the difficulty that students may encounter while solving them. Such prior knowledge about questions' complexity should help instructors assign questions the proper points and place them in a progressive order throughout the exam leading to a more reliable evaluation tool.**

Keywords*— assessment; code metrics; novice code writing questions; question difficulty.

I. INTRODUCTION

Requests to write novice code can happen at many occasions. For example, junior college students are typically asked to write simple programs. IT Job interviews often involve some writing of code fragments. Even programming contests start off with relatively simple programs writing. Finally, novice programming questions are commonly used tools to assess program outcomes in some academic accreditation bodies such as ABET [1].

A large body of studies in the literature [2] confirm that assessing the difficulty level of such code writing tasks or questions is not easy nor accurate. The low passing rate of students in their first year of coding is not uncommon. This might be partially explained by the lack of good assessment tools for ranking the difficulty of code writing questions. Researchers such as Whalley et al. [3] reported that assessing programming questions fairly and reliably is a challenging process, which brings the need for tools and frameworks that can help in that regard. Elliott Tew [4] drew the attention to the lack of reliable assessment tools in the field of computing, in teaching and research. Whalley et al. [5] argued that there is a real need for more effective assessment tools for novice programmers in order to enhance their learning experience.

Our research question in this work can be summarized as follows: given a simple code writing question and its sample answer (given by the instructor, for example), how difficult will it be for students to answer this question. Our approach depends on developing a predictor of the difficulty of a programming question by observing the correlation between the static properties of the sample code and the observed student performance. The scope of our work is not concerned with assessing the difficulty of code reading tasks such as the tracing questions we typically see in the exams of programming courses. Rather, we focus only on the difficulty that the learner may face to get the right answer for a question that require code writing.

II. BACKGROUND

In this section we review the two commonly used approaches to assess the difficulty of coding questions in general, namely taxonomies and metrics. The *taxonomies* approach is based on the premise that the computing instructors, as the creators of the coding questions, are the most knowledgeable persons about how challenging their questions are. Therefore, we may trust the labels (e.g., easy, hard, challenging) they affix to each question. This approach is subjective as these labels may vary from one instructor to another. The other approach, *metrics*, is a quantitative approach where we count on analytical tools that analyze the sample answer for the coding question and returns a numerical value that help rank the difficulty of the question under investigation. Next, we elucidate the two approaches and how they are used.

A. Taxonomies: The Cognitive Approach - Why They Are Not the Answer

An intuitive solution for assessing the difficulty of a coding question is to view it as a classification problem, i.e., classifying the difficulty of a given code writing question based on its instructor-supplied sample solution. This approach compelled computer science educators to think of using pedagogical taxonomies such as Bloom's [6], the revised Bloom's [7] and the SOLO Taxonomy [8].

Bloom's taxonomy helps classify the difficulty of the code at the level of thinking. The revised Bloom's taxonomy helps do this classification based on the level of knowledge or skill needed to solve a question. Using these taxonomies in practice have some challenges such as the difficulty of interpretation [9] [10][11] and the need for profound understanding of the context of the problem at hand.

978-1-4673-8633-3/16/$31.00 ©2016 IEEE 10-13 April 2016, Abu Dhabi, UAE
2016 IEEE Global Engineering Education Conference (EDUCON)

Furthermore, it is not easy to get a consent in case we have multiple classifiers where each may have a preconceived understanding of the complexity of the problem and the cognitive level and skills needed to solve it by students [12].

SOLO is another taxonomy that captured the attention of many researchers and educators for the purpose of both reading and writing coding questions. For example, some researchers [13][14] found that SOLO can be used with code comprehension questions if the classifiers share the understanding of the context and application. For code writing, Lister et al. [15] introduced some guidelines in order to tag code writing questions with the SOLO levels. However, even with these guidelines, the process seemed difficult [11].

Some researchers augmented the SOLO taxonomy with another tool. For example, Meerbaum-Salant, Armoni and Ben-Ari [16] chose to combine SOLO with Bloom's taxonomy. Another attempt was done by Whalley et al. [5] who used a framework of salient elements, applied to paper-based programming examinations, in order to clarify the definition of the SOLO categories.

In general, whether assessing code reading or writing questions, numerous studies [13][14][5] confirm that there is a conformance between the difficulty of the question, measured by the performance of students, and the SOLO level assigned to it – the higher the SOLO level, the more difficult the question is.

Despite the progress made towards estimating the difficulty of comprehension questions, Simon et al. [17] stressed on the need for a more reliable means of assessing the difficulty of code writing questions, especially at higher levels of details than what taxonomies offer. One of these recent attempts is by Whalley and Kasto [18] where they used some software metrics to evaluate the complexity of writing novice code. However, we found that

1) Their metrics selection is not justified well.
2) The metrics could be weighted
3) The used data may have some anomalies. For example, all questions were inter-related. The questions adhered to a progressive scenario that aims, eventually, to solve a large problem. This latent interdependency of questions usually appear in course projects, not in exams. We avert these pitfalls in our experimentation data by using independent exam questions and by applying them to a bigger student population.

B. Metrics: The Systematic Approach

A less subjective approach is to derive metrics from a given text of code in order to assess its complexity or difficulty. For the purpose of measuring the readability of code, Starsinic [19] developed a Perl script that adopts the Flesch-Kincaid readability metric [20] used with English prose in order to gauge the readability of simple programs. Börstler, Caspersen and Nordström [21] carried out a similar research where they developed a framework that used a combination of 3 metrics in order classify whether a code sample is good (readable and comprehensible) or bad. These 3 metrics are: the cylcomatic complexity [22], the Software Readability Ease Score (SRES) which is an interpretation of the English language Flesch Readability Ease Measure [23] and Halstead's difficulty metric [24].

Kasto and Whalley [25] used metrics to assess two types of code questions: code tracing and Explain in Plain English (EipE). They tried to explore the relationship between the metrics and the student performance, represented by the percentage of fully correct answers. To assess the complexity of code tracing questions, the authors used dynamic metrics, along with cylcomatic complexity and average block depth, and found significant correlation between them and the difficulty of the question. They also investigated the use of metrics for the Explain-in-plain-English (EipE) questions but did not find any significant correlations between difficulty and Halstead metrics or the cyclomatic complexity.

In this paper we report on our attempts to use software metrics solely as a tool for estimating the difficulty of novice code writing tasks.

III. Our Approach: Software Metrics

Our aim is to assess the difficulty of a code writing question, i.e., to estimate the complexity of code that is yet to be written by students. To that end, our approach relies on the premise of using the characteristics of the sample solution to predict the performance of students on that particular question. These characteristics are represented by software metrics that are directly derived from the syntax of the sample code.

As a mental process, and regardless the final quality produced, writing a program entails structuring the code in some way or another. The complexity of this structure may reflect the cognitive effort (difficulty) needed to solve the problem. Next, we list the structure-relevant metrics that we believe to be useful for predicting the difficulty of a code writing task:

- *Cyclomatic Complexity (M1)*: It is a quantitative measure of the number of linearly independent paths, including decisions, through the source code [22]. This includes modern constructs such as the try-catch-finally construct.
- *Average Depth of Nested Blocks (M2)*.
- *The total number of commands (or statements) (M3)*: it is the number of Java method calls. Initially, we thought of considering the number of lines of code but this may cause inaccuracy due to the excessive use, or omission, of block brackets and due to the different code formatting styles adopted by different users.
- *The total number of operators (M4)*.
- *The number of unique operators (M5)*.

It is worth mentioning that Kasto and Whalley [25] introduced two dynamic metrics in their work:

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

TABLE I. THE RDI VS. SOFTWARE METRICS FOR THE MODEL SOLUTION OF EACH QUESTION

Questions Ranked Based on RDI	1	2	3	4	5	6	7	8	9	10
Real Difficulty Index (RDI %)	**0**	**6**	**10**	**16**	**21**	**37**	**45**	**48**	**61**	**67**
Cyclomatic complexity (M1)	1	2	2	3	4	4	5	8	8	9
Average nested block depth (M2)	1	2	2	2	2	3	4	5	5	6
Number of operators (M3)	1	1	3	4	8	6	7	11	12	14
Number of unique operators (M4)	1	1	3	4	5	5	6	7	8	8
Number of commands (M5)	4	7	7	7	10	12	13	16	17	19
Predicted difficulty Index	**8**	**13**	**17**	**20**	**29**	**30**	**35**	**47**	**50**	**56**

TABLE II. THE CORRELATIONS BETWEEN CODE METRICS AND THE DIFFICULTY OF WRITING THAT CODE

Software Metric	**Pearson's Correlation**	
	r	***p***
Cyclomatic complexity (M1)	0.95973	1.1E-05
Average nested block depth (M2)	0.968277	4.26E-06
Number of operators (M3)	0.941469	4.78E-05
Number of unique operators (M4)	0.95917	1.16E-05
Number of commands (M5)	0.984828	2.28E-07
Predicted difficulty Index	0.978496	9.12E-07

However, we argue that these metrics are applicable to code tracing questions only. In such questions, only the paths of code that the students must trace through can add to the complexity. Since we are concerned with code writing only, we contend that these dynamic metrics are irrelevant to our work.

Nowadays, many software tools are available that can compute code metrics automatically for many programming languages [29]. A few examples to list are Radon [26], Microsoft Visual Studio [27], and IBM Rational Software Analyzer [28]. Given that we are investigating novice code writing questions, analyzing the code metrics for intrinsically small model solutions in our experiments was easy to analyze manually without the need for any of the software tools listed above.

IV. DATA AND EXPERIMENTAL SETUP

A. Identifying Code Metrics that Can be Used as Predictors

The core of our solution relies on investigating the correlation between the difficulty of a code writing question and the software metrics of its model answer. To that end, we undertake the following steps:

a) Collect a sample of code writing questions with different shades of difficulty levels. The difficulty of each question is determined by the student performance measured by the percentage of students who failed to earn at least 70% of the maximum mark on that question. We call this measure the *Real Difficulty Index (RDI)*. The sampled questions are "unseen", meaning, they either involve new code syntax/language constructs, or a variation of code that was seen through the course. For example, asking to find the largest number in an array after they have learned to find the lowest in class.

b) Rank the above questions based on the Real Difficulty Index.

c) Compute the code metrics for each question in the model solution.

d) Compute the Pearson correlation between the RDI and each software metric.

e) Based on the computed correlations in (d) we decide on which software metrics can be reasonably used as predictors for the difficulty of a given question. More specifically, we need to select:

- Which metrics that have strong correlation with the RDI.
- Whether we need to assign different weights for each metric, if needed.
- How to combine the 5 metrics in order to device one predictive index through which we can infer the RDI.

The coding questions we used are collected from an introductory programming course taught in Java. Fortunately, we maintain detailed information about the performance of each student under each question in an exam/quiz because we had to track such data using a repository of Excel sheets for the purpose of ABET course assessments in all semesters.

First, we sampled 10 questions ranked based on the Real Difficulty Index. Appendix A lists these questions where Question 1 is deemed the easiest while Question 10 is the hardest. The questions are extracted from different exams and quizzes but all were undertaken by the same student population who were taught by the same instructor.

Then we manually analyzed the characteristics of the solution code of these questions based on the 5 software metrics listed above. Table I lists the value of each software metric for each question.

TABLE III. THE PDI-BASED RANKING IS CLOSE TO THE RDI-BASED RANKING

Question #	1	2	3	4	5	6	7	8	9	10
RDI	8	9	10	14	32	49	50	54	66	67
PDI	11	14	***12***	27	29	40	***38***	57	56	62

TABLE IV. RDI AND PDI FOR A RECURSION-BASED VS. LOOP-BASED IMPLEMENTATION

	Q11 (Recursion)	Q12 (Loop)
RDI	49%	19%
PDI	15	21

Table II summarizes the Pearson correlation between the RDI and the 5 code metrics. We can see that strong correlations exist between the 5 code metrics and the Real Difficulty Index of writing such code by students. The Pearson correlation coefficients for all metrics are near +1, indicating strong correlations backed up by the low *p*-values associated with these coefficients (less than α=0.001). Additionally, we computed the Multiple R correlation coefficient, which has a range of [0, 1], in order to measure the strength of the correlation between the RDI and the code metrics collectively. The observed Multiple R value (0.993197, almost 1) again confirmed the strength of this correlation.

This may suggest that code metrics can be exploited for predicting the performance of students for a given code writing question. Since all metrics showed strong correlations, we introduce a single metric, we call it the *Predicted Difficulty Index (PDI)*, which linearly combines all metrics using the following formula:

$$\textit{Predicted Difficulty Index} = M1 + M2 + M3 + M4 + M5$$

Since all metrics equally showed strong correlation, we trust that we do not need to assign different weights to the metrics M1 to M5 in the above equation. The last row of Table I shows the PDI values for each question. Questions can be ranked based on these PDI values in order to infer their original ranking that could be based on the Real Difficulty Index.

B. Evaluating the PDI Predictor

To test the effectiveness of our Predicted Difficulty Index as a predictor for the difficulty of questions, we carried out the following procedure:

a) Sampled 10 other questions.

b) Ranked these questions based on their RDI values.

c) Computed the PDI values for these questions.

d) Ranked the questions based on their PDI values.

e) Compared the two rankings in (b) and (d).

Table III summarizes the RDI and PDI values for the test set of 10 questions. Originally, we sorted the questions based on the RDI values. By contrasting this RDI-based ranking to the PDI-based one, we can observe that only questions 3 and 7 fail to conform to the RDI-based ranking. We can conclude that the PDI-based ranking can be generally useful for the purpose of ranking questions and differentiating their relative difficultly.

C. Limitation and Special Cases

All selected questions above focused on solutions that involve traditional programming control flow (e.g. sequential statements, if-else, or loop statements such as while and for). We got curious to observe the correlation between the RDI and the PDI for questions that involve special programming paradigms such as recursion. To that end, we selected two questions:

- Q11: A question that asks students to find the sum of numbers from 1 to N using recursion.
- Q12: The same as Q11 but students should use loops instead of recursion.

As shown in Table IV, the RDI of Q11 (recursive question) falls between the difficulty of Q8 and Q9 (see Table I), i.e., it is relatively hard to solve. Nevertheless, the PDI ranked its difficulty to be between the difficulty of questions Q2 and Q3, i.e., easy to solve, which is a big misprediction. Interestingly, Q12, which achieves the same goal of Q11 but using a classic loop-based implementation, is ranked to be as difficult as Q4 to Q5. This is confirmed by the agreement between the rankings of the questions' RDI and our PDI prediction tool.

Apparently relying solely on the code structure of questions that involve special programming mental models such as recursion is not sufficient as these cognitive models seem to impose some latent difficulty for students.

V. ASSUMPTIONS AND LIMITATIONS

It is worth highlighting the assumptions and limitations surrounding our work, which can be summarized as follows:

- Metrics are derived from the sample solution provided by the instructor. We realize that such a solution may unintentionally be less complex than solutions given by students. In other words, the code provided by a student can be an acceptable solution but does not match the model solution provided by the instructor at the cognitive level. For example, the instructor's code could be extra optimized.
- We claim that our approach is valid for novice programming questions, not for large-scale projects.
- The inclusion of readability metrics is based on the hypothesis that a less readable code is usually more difficult to write too.
- Our solution is not meant to provide an absolute index of difficulty, rather it is an indicative index that help rank (sort) questions based on their challenging nature.
- `Our solution may not be suitable for special programming paradigms such as recursion. Recursion as a concept entails its own cognitive and mental requirements on students when asked to write the solution. For example, we noticed that students with prior programming knowledge may prefer a loop-based solution to a recursion-based one for the same given problem. In the future, we will try to tackle this limitation by augmenting our metrics-only predictor with a cognitive tagging mechanism such as the SOLO taxonomy [8].

VI. Conclusion

Designing a good exam that involves code writing questions requires that exam designers, e.g., college instructors, have a prior knowledge of each question's level of difficulty or at least the ability to rank questions according to the difficulty that students may encounter as they answer them. In this work, we hypothesized that the code metrics directly derived from code structure can be used as predictors for the underneath difficulty that student may face and eventually be reflected on their marks.

To that end, we investigated the existence of correlation between the real difficulty index, measured by student performance, and the five code metrics that are reasonably relevant. Based on real exam questions, we found that there is strong correlations between the Real Difficulty Index (%) and the code metrics proposed. As a result, we introduced the Predicted Difficulty Index which is a linear combination of all code metrics. As a result, the PDI can be useful to rank a given set of questions based on their anticipated difficulty. We validated the PDI's feasibility on an independent set of questions and it satisfactorily succeeded to rank these questions based on the observed empirical difficulty represented by the student performance (measured by the Real Difficulty Index).

We highlighted the assumptions associated with our work and the limitations of our approach. An example of the latter is handling special programming concepts such as recursion. We showed that such situations we cannot solely rely on metrics derived from code structures in order to anticipate the difficulty of answering such questions by students.

From a practical point of view, exam designers and educators can tap into our research outcome by ranking their coding questions based on the Predicted Difficulty Index and consequently assign the appropriate rewarding points to each question based on this ranking. Another benefit from our research outcome is the ability to tune the difficulty level of a coding question by increasing or decreasing the values of the 5 software metrics we identified in this work. This should generally lead to better exam designs and assessment tools.

VII. Future Work

For future work, we believe that the following challenges create research opportunities:

- The Predicted Difficulty Index presented in this paper works at the syntactic level of the code and is oblivious to the code semantics that may embed intricate concepts such as recursion. As we showed, this can spoil the accuracy of the PDI and render it less useful. To redress this shortcoming, we believe that we need to take into account factors such as the level of thinking required and the cognitive load imposed on students by a given question. This concern was confirmed by our empirical experience and observation when we experimented with the recursion question. Despite the simplicity of the solution from the perspective of code structure, the Real Difficulty Index is still high. This might be attributed to the cognitive requirements that such type of programming questions may entail. In a future research, we may augment the code metrics with another dimension that addresses the cognitive level each question may require by using one of the common educational taxonomies such as SOLO [8].
- It is intriguing to investigate the feasibility of our solution with other programming paradigms such as object-oriented programming and functional programming.
- We claimed that our approach is feasible with novice code writing questions. It is interesting to explore if this is still valid and applicable to medium-to-large programming projects.
- It is interesting to implement our methodology by having a tool that accepts a set of model answers for programming questions, and outputs them ranked based on the Predicted Difficulty Index. This tool should help educators, for example, assign the proper marks for reach question, leading to a better exam design and assessment.

References

[1] www.abet.org, 'ABET', 2016. [Online]. Available: http://www.abet.org. [Accessed: 01- Jan- 2016].

[2] A. Robins, J. Rountree and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, 13(2): 137-172, 2003.

[3] J. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, and C. Prasad, "An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies," *Australian Computer Science Communications*, 52: 243-252, 2006.

[4] A. Elliott Tew, "Assessing fundamental introductory computing concept knowledge in a language independent manner," PhD dissertation, Georgia Institute of Technology, USA, 2010.

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

[6] B. S. Bloom, *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*, Addison Wesley, 1956.

[7] L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, et al., *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*, Longman, 2001.

[8] J. B. Biggs and K. F. Collis, *Evaluating the Quality of Learning: The SOLO Taxonomy*, New York, Academic Press, 1982.

[9] U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, et al., "Developing a Computer Science-specific Learning Taxonomy," *ACM SIGCSE Bulletin*, 39 (4):152-170, 2007.

[10] E. Thompson, A. Luxton-Reilly, J. Whalley, M. Hu, and P. Robbins, "Bloom's Taxonomy for CS assessment," *Conferences in Research and Practice in Information Technology*, 78: 155-162, 2008.

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

[12] R. Gluga, J. Kay, R. Lister, S. Kleitman, and T. Lever, "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, 2012.

[13] T. Clear, J. Whalley, R. Lister, A. Carbone, M. Hu, J. Sheard, et al., "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, 2008.

[14] J. Sheard, A. Carbone, R. Lister, B. Simon, E. Thompson, and J. L. Whalley, "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, 2008.

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

[16] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, *Learning Computer Science Concepts with of the Observed Learning Outcome,* New York, 2010.

[17] B. Simon, M. Lopez, K. Sutton, and T. Clear, "Surely we must learn to read before we learn to write!," *Conferences in Research and Practice in Information Technology*, 95: 165-170, 2009.

[18] J. Whalley, and N. Kasto, "How difficult are novice code writing tasks? A software metrics approach," *Proc. Sixteenth Australasian Computing Education Conference (ACE2014)*, Auckland, New Zealand. CRPIT, 148. Whalley, J. and D\'Souza, D. Eds., ACS. 105-112, 2014.

[19] K. Starsinic, "Perl Style," *The Perl Journal*, 3(3), Fall 1998.

[20] J. P. Kincaid, R. P. Jr. Fishburne, R. L. Rogers, and B. S. Chissom, "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, 1975.

[21] J. Börstler, M.E. Caspersen, and M. Nordström, "Beauty and the Beast — Toward a Measurement Framework for Example Program Quality," *Technical Report*, Department of Computing Science, Umeå University, ISSN 0348-0542, 2007.

[22] T. McCabe, "A Software Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2(4):308-320, 1976.

[23] R. Flesch, "A new readability yardstick," *Journal of Applied Psychology*, 32: 221-233, 1948.

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

[25] N. Kasto and J. Whalley, "Measuring the difficulty of code comprehension tasks using software metrics," *Proc of the 15th Australasian Computer Education Conference (ACE 2013)*, Adelaide, Australia, 57-6, 2013.

[26] radon.readthedocs.org , Radon, 2016. [Online]. Available: http://radon.readthedocs.org/en/latest/intro.html. [Accessed: 01-Jan-2016].

[27] msdn.microsoft.com, Microsoft VisualStudio, 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb385914.aspx. [Accessed: 01-Jan-2016].

[28] www.ibm.com, IBM, 2016. [Online]. Available: http://www-03.ibm.com/software/products/en/ratisoftanalfami. [Accessed: 01-Jan-2016].

[29] en.wikipedia.org , Wikipedia - Code Analyzers. [Online]. Available: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis. [Accessed: 01-Jan-2016].

APPENDIX A: CODING QUESTIONS

Q1. Write a program that asks the user to enter her name. The program should greet the user by this name.

Q2. Write a program to determine whether a person is adult (18+) based on the input age.

Q3. Write a program to convert euros to dollars. The input euros must be >=0.

Q4. Write a program to convert a Celsius temperature to Fahrenheit. The Celsius temperature must be in the range [-50, 50].

Q5. Write a program that allows the user to convert Celsius to Fahrenheit and vice versa.

Q6. Write a program that asks the user to enter a number between 1 and 10 and displays the multiplication table for that number.

Q7. Write a program to display the times table of a number X entered by the user. X is between 1 and 10. The program continues until the user opts to stop.

Q8. Write a program to display the numbers that are divisible by 5 in the range [n1, n2], where n1 and n2 are entered by the user. The program continues until the user opts to stop.

Q9. Write a program to display all numbers in the range [n1, n2] whose squares are divisible by 2, where n1 and n2 are entered by the user. The program continues until the user opts to stop.

Q10. Write a program to display the factorial of any number X that is divisible by 10. X is an integer in the range [n1, n2] where n1 and n2 are entered by the user. The program continues until the user opts to stop.

Q11. Write a recursive method that returns the sum of numbers from 1 to N.

Q12. Write a loop-based method that returns the sum of numbers from 1 to N.