Promoting Code Quality via Automated Feedback on Student Submissions

Oscar Karnalim*† and Simon*
*School of Electrical Engineering and Computing, University of Newcastle, Australia
oscar.karnalim@uon.edu.au, simon@newcastle.edu.au
†Faculty of Information Technology, Maranatha Christian University, Indonesia

***Abstract*—This research-to-practice work-in-progress paper presents an automated feedback tool that can be used in many teaching environments by integrating it with a web-based assessment submission system. Each time a student submits their work, they will automatically get feedback about aspects of the code quality. Automated feedback tools have been developed to educate students about code quality. However, integrating such a tool into an existing teaching environment can be challenging as these tools can depend on particular working environments, can be separate from the assessment submission system, or can require historical data. Our initial evaluation shows that the tool can be helpful as students do sometimes neglect to satisfy all code quality requirements. However, some false results are expected for spelling correction as student programs are not written in natural language. According to our quasi-experiments, the tool substantially reduces the number of word misspellings in comments due to their substantial frequency of occurrence.**

***Index Terms*—code quality, automated feedback, assessment, programming, computing education**

I. Introduction

Code quality is identified by the clarity of the code to be understood without the need to test the code or read its documentation [1]. This is arguably crucial for software developers [2] as they are expected to write a high volume of code while collaborating with one another. Although code quality is a serious concern in industry, it is not the primary focus in academia [3], [4]. On most occasions, programming assessment is dedicated to measuring student' ability to solve problems and/or implement algorithms. Further, code quality is often not covered in course materials. Some instructors do consider it in their grading process, but as the weighting is generally small compared with other tasks, this might not be sufficient to motivate students to learn about writing high-quality code.

Some automated feedback tools have been developed to help students learn for themselves how code quality can be improved. Blau and Moss [5] propose a plugin for Eclipse – a programming working environment – for students who are familiar with Java, dealing with misuses of fields, public modifiers, booleans, and loop control. Keuning et al. [6] developed a web-based tool, specifically for students with prior Java programming knowledge, which provides step-by-step instructions. A tool introduced by Ureel and Wallace [7] provides the feedback while students are writing their programs. Choudbury et al. [8] propose a tool that can automatically measure code quality based on historical data.

Nevertheless, integrating such a tool into an existing teaching environment can be challenging as these tools either depend on a particular working environment (e.g., Eclipse), are separate from the assessment submission system, or require historical data.

This paper proposes code clarity suggester (CCS), an automated feedback tool that can be used in many teaching environments. The tool is integrated in a web-based assessment submission system where each assessment task will be assigned a unique submission link that can be embedded in many learning management systems. Each time a student submits their work, they will automatically get feedback about the code quality in a HTML page whose URL is emailed to the student. For simplicity, students can log in while submitting the code. Further, the feedback is accessible without login as the URL is unique and sent privately by email.

At this preliminary stage, the tool provides feedback at documentation level (identifier names, header comments, and body comments [1]) for Python and Java submissions. Specifically, it will suggest that students correct misspelled words, rename identifiers, rewrite comments that are either too short or not meaningful, ensure that all identifiers consistently satisfy a particular naming style, and check that each code block has a comment.

II. The Tool

CCS works in four stages for each program submitted. First, the submission is converted to a token string using ANTLR [9] (a code parser) while preserving its comments and white space.

Second, the tool detects all code blocks and suggests adding a header comment for each code block that lacks one (we will call this NHC, for no header comment). A code block is defined as a group of syntax tokens that have no blank lines among them. One or more blank lines are treated as block delimiters, following the algorithm used in a similarity disguiser to add a comment for each code block [10].

Distinct identifiers are then filtered and further processed. Suggestions are generated for identifiers that are too short (SI) – i.e., no more than three characters, inspired by early information retrieval systems that ignore such words [11].

For each identifier, its sub-words are then extracted and checked for correctness and meaning. The extraction process, adapted from a software search engine [12], is based on two styles of concatenating identifier sub-words: camel case (e.g., 'thisIsAVariable') and underscore (e.g., 'this_is_a_variable'). The extraction also deals with some special cases. For example, 'IEEE' in 'IEEEVariableToShowData' will be treated as one sub-word, not four. All sub-words are then lowercased and suggestions are generated for identifiers without any meaningful sub-words (which we will call MI), identifiers with misspelled sub-words (MSI), and identifiers with inconsistent concatenation style (ICI).

Meaningful sub-words should consist only of letters and should not be common. The common word list is adapted from Apache Lucene's stop words[1], depending on the language chosen, either English or Indonesian. Apache Lucene is also used to detect misspelled sub-words and provide their closest correct forms in the suggestion. The detection is based on three external dictionaries, two for English (American[2] and British[3]) and one for Indonesian[4].

Identifiers with inconsistent concatenation style are detected in two stages. First, camel-case-delimited identifiers and underscore-delimited identifiers are counted separately. The majority style is the one with more identifiers. After that, all identifiers whose concatenation style is different from the majority style are marked as inconsistent.

Some identifiers are quasi-keywords from third-party libraries and it is not appropriate to suggest improvements in their name quality. CCS can remove those identifiers from consideration if they are listed as additional keywords. This follows the mechanism used by a similarity disguiser, which does not disguise such identifiers [10].

In the fourth stage, comments are filtered and suggestions are generated for any comments that are too short (SC), have no meaningful words (MC), or have misspelled words (MSC). The detection mechanisms are somewhat similar to those used for identifiers, but with different justifications. Short comments are those whose length is less than seven characters for Java or six characters for Python, as three is the minimum limit of considered words in early information retrieval systems [11] and the rest is the length of the comment prefix for Java or Python plus one space. Comment words are extracted by treating non-alphanumeric characters as delimiters and then extracting the sub-words of all identifier-like words.

For reusability purposes, the tool is designed as a library so that it can be integrated into larger systems. Given a program, and arguments set via command-line instructions, the tool will generate code quality feedback in a text file for further processing. The result can also be visualized in an interactive HTML page as shown in Fig. 1. The general guideline panel displays tips to improve code quality. The submitted code panel shows the submitted program formatted with Google Prettify[5]; highlighted tokens refer to approximate locations where code quality can be improved. The suggestion list panel lists all code quality suggestions for the submitted program. Clicking a suggestion or its highlighted token will show the detail in the detail panel.

For each unique identifier in the code, its suggestion (if any) will be assigned only to the first occurrence, to avoid unnecessary repetition. For each NHC suggestion (a comment for each code block), the highlighted token will be the first one of the corresponding code block. Suggestions will be merged if they refer to the same highlighted token.

Anyone who would like to use the library can download it from our GitHub repository[6]. They can also modify and upgrade the code according to their needs.

Currently, CCS is integrated into an unpublished assessment submission system that sends a code quality report for each submitted program via email. Students are expected to learn from the report and improve their code quality for future submissions.

III. Initial Evaluation

The introduction of CCS gives rise to three initial research questions:

RQ1 Is there a role for the tool when the students are already encouraged to write clear code?

RQ2 What are the expected false results of spelling corrections (MSI and MSC suggestions)?

RQ3 How effective is the tool to educate students about code quality?

RQ1 was addressed by measuring the proportion of low-quality code fragments in programming assessments in a course where high code quality is encouraged. The tool is considered to be worthwhile if the proportion is occasionally substantial. Considering the eight code quality aspects of CCS (NHC, SI, MI, MSI, SC, MC, MSC, and ICI), the proportion of each was measured by considering only relevant code fragments. Code aspects relating to identifiers (SI, MI, MSI) only consider distinct identifiers, while code aspects about word misspelling (MSI and MSC) rely on sub-words rather than code fragments.

The data set covers sixteen assessment tasks, with a total of 745 student programs, from three offerings of an introductory programming course in which high code quality is encouraged. The numbers of programs can be seen in Table I. Campus ID is a code for the campus where the course was offered. For each offering, two assessment tasks were given, to be completed with Python and JES[7], a media computation library for introductory programming. Standard Python and JES library keywords that are recognized as identifiers were excluded from consideration.

RQ2 was addressed by manually observing corrected spellings for both identifiers and comments in the same data

[1]https://lucene.apache.org/

[2]https://github.com/dwyl/english-words

[3]https://www.curlewcommunications.uk/wordlist.html

[4]http://indodic.com/SpellCheckInstall.html

[5]https://github.com/google/code-prettify

[6]https://github.com/oscarkarnalim/ccs

[7]http://coweb.cc.gatech.edu/mediaComp-teach

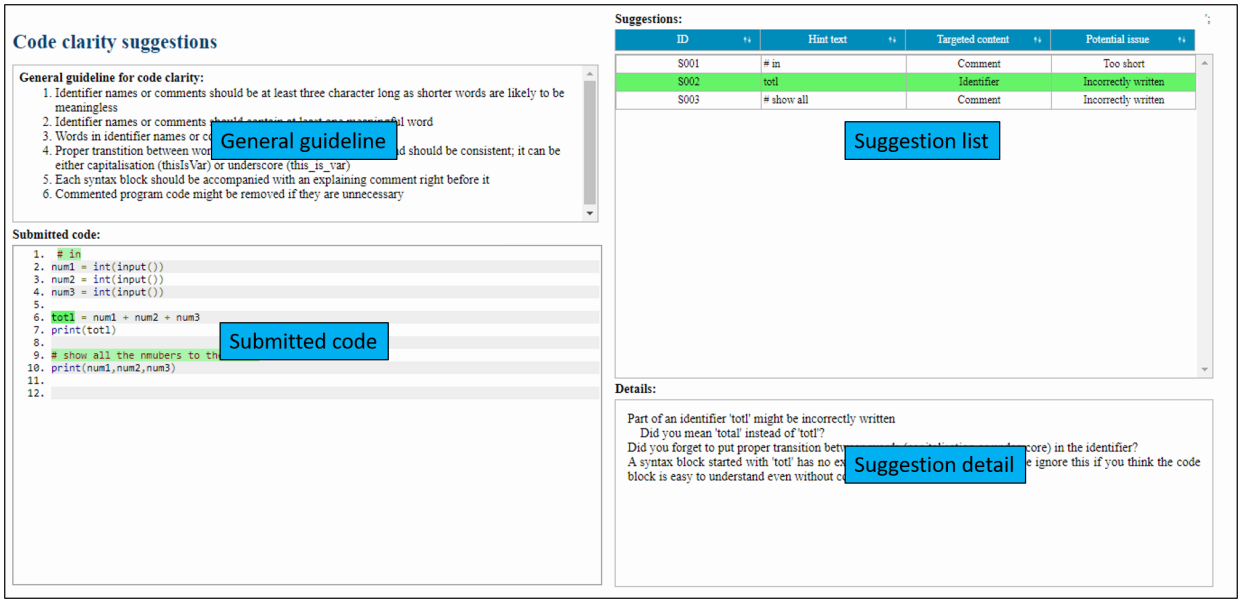

Fig. 1. CCS interactive HTML page, illustrating the general layout

TABLE I
STUDENT PROGRAMS IN INTRODUCTORY PROGRAMMING DATA SET

Campus ID	Offering	1^{st} assessment	2^{nd} assessment
C	2018	133	89
O	2018	25	15
S	2018	32	5
C	2019	92	67
O	2019	29	16
S	2019	10	6
C	2020	132	67
O	2020	20	7

set. Although many of the corrections make sense, about one in four is a false result, a correctly spelled word that is considered as misspelled. All false results were grouped based on their shared characteristics and summarized.

RQ3 was addressed by performing two quasi-experiments with students from another university. The first experiment involves two offerings of introductory programming to information systems (IS) students: a control group (26 students) from 2019 and an intervention group (33 students) from 2020. The second experiment involves two different information technology (IT) courses: introductory programming (61 students) in 2019 and advanced algorithms and data structures (43 students) in 2020. These include many of the same students as they are compulsory courses for year 1 and year 2. The latter makes substantial use of template code, which was removed with common code segment removal [13] to make the assessments comparable with those of the former.

All programming courses described here have comparable assessment design. Each week, students need to finish both lab and homework assessments. The former are to be completed in a two-hour class, while the latter can be submitted any time before the start of the following week's class. For richer analysis, we separate lab and homework assessments.

For both intervention groups, CCS was exclusively used starting from week 4, with the preceding weeks used to measure natural differences between the control and intervention groups. Each time a student submitted a program, code quality feedback was generated and the link was sent to the student's email. The feedback was in student's native language, which is Indonesian. However, spelling correction was based on the combination of English and Indonesian as both are often used in identifiers and comments.

CCS is argued to be more effective if its usage substantially reduces the proportion of low quality code fragments from week 4 onward. If the difference is also found in the first three weeks, it is a natural difference rather than an impact of the tool. A two-tailed t-test was used to test the significance, with 95% confidence rate.

A. *Addressing RQ1: The Need for CCS*

Table II shows the highest proportion of low quality for each code aspect, with assessment ID as the concatenation of assessment number, offering year, and campus initial. On more than half of the targeted code aspects, the highest proportion is quite substantial. Further, the highest proportion varies with assessment task. CCS still has a role as a reminder since some students might occasionally neglect to follow the guidelines.

NHC (no header comment before syntax block) has the highest proportion since students were encouraged only to put comments on complex and/or long syntax blocks, arguing that the rest can be self-explanatory without comments. Further, for function headers, they were encouraged to put comments after the header rather than before it. In these cases, CCS can remind students to check whether the syntax blocks are self-explanatory, are part of function headers, or should be suitably commented.

TABLE II
HIGHEST PROPORTION OF LOW CODE QUALITY ASPECT

Code aspect	Highest proportion	Assessment ID
NHC	64%	1_2019_C
SI	16%	1_2019_S
MI	7%	1_2019_S
MSI	16%	1_2018_S
SC	10%	1_2020_C
MC	3%	1_2020_O
MSC	4%	1_2020_C
ICI	2%	2_2020_C

TABLE III
SIGNIFICANT CHANGES IN PROPORTION OF LOW-QUALITY CODE FRAGMENTS FROM WEEK 4 ONWARD

Code aspect	Assessment ID	Control	Intervention	p-value
MI	IS_Lab	<1%	2%	<0.001
	IT_Lab	2%	<1%	<0.001
	IT_Homework	2%	<1%	<0.001
MSI	IT_Lab	16%	23%	<0.01
SC	IS_Homework	<1%	2%	0.02
MC	IS_Lab	<1%	2%	0.01
	IS_Homework	<1%	3%	<0.001
	IT_Homework	<1%	<1%	<0.001
MSC	IS_Lab	35%	28%	<0.01
	IT_Homework	37%	25%	<0.001
ICI	IS_Homework	<1%	1%	<0.01

B. Addressing RQ2: Expected False Results of Spelling Corrections

For identifiers, the false results cover abbreviations (e.g., 'kps' for kilometers per second), contractions (e.g., 'nums' for numbers), concatenated words without delimiters (e.g., 'datapoints' for data points), programming terminologies (e.g., 'todo' for something that needs to be done), and combinations of them (e.g., 'targ' for target argument). For comments, the false results are similar since identifiers are often used in comments. However, they also include 'names' [14] such as student names (used for personal information), website domains (used for referencing outsourced code), and programming products such as Jython (used for describing third-party libraries). Although these terms are self-explanatory to the program's author, explanations might nevertheless be helpful for other programmers, so CCS reminds the author to at least check the clarity of these terms.

C. Addressing RQ3: Effectiveness of the Tool

Table III shows significant changes exclusive to week 4 onward, with assessment ID as a concatenation of major and assessment type. CCS significantly affects some code quality aspects, but the only substantial ones are those related to word misspellings (MSI and MSC), as the proportion of other code quality aspects is already low.

The tool seems to increase the proportion of identifier word misspelling (MSI) in one case since the assessments of the intervention group tend to expect many identifiers with more than one sub-word. Students tended to name such identifiers by concatenating sub-words without using either camel case or underscores; since that is undetectable by the tool, it results in more identifier word misspellings.

In comments, the proportion of word misspellings (MSC) is substantially reduced in two cases. The tool accurately helps students to find such misspellings as most of the words are listed in the dictionary and comment words are generally separated by spaces and punctuation.

IV. CONCLUSION AND FUTURE WORK

This paper proposes an automated feedback tool to educate students about some aspects of code quality, which can be used in many teaching environments. Our observation shows that the tool can be useful, as students are not satisfying all code quality requirements due to human error. While using the tool for spelling correction, false results can occur, covering abbreviations, contractions, concatenated words without delimiters, programming terminologies, named entities, and combinations of those. According to our quasi-experiments, the tool only substantially reduces the number of comment word misspellings, as those occurred frequently and were detected accurately.

The research is not yet complete. We plan to include other aspects of code quality to further educate students in those aspects. Examples are suggesting the replacement of inefficient code fragments, suggesting the removal of unused variables or functions, checking whether the layout is readable, and pointing out possible logic and runtime error. We also intend to reconduct the quasi-experiments on courses with larger assessments since the chance of human error is typically higher in longer programs.

ACKNOWLEDGMENT

To Australia Awards Scholarship for supporting the study of the first author. To Gisela Kurniawati and Sendy Ferdian Sujadi from Maranatha Christian University, Indonesia for facilitating the quasi-experiments in their courses. To William Chivers from University of Newcastle, Australia, for his contribution to the overall project.

REFERENCES

[1] M. Stegeman, E. Barendsen, and S. Smetsers, "Towards an empirically validated model for assessment of code quality," in *14th Koli Calling International Conference on Computing Education Research*. ACM, 2014, pp. 99–108.

[2] J. Börstler, H. Störrle, D. Toll, J. van Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, and B. MacKellar, ""I know it When I see it": perceptions of code quality," in *ITiCSE 2017 Working Group Reports*. ACM, 2018, pp. 70–85.

[3] D. Kirk, T. Crow, A. Luxton-Reilly, and E. Tempero, "On assuring learning about code quality," in *22nd Australasian Computing Education Conference*. ACM, 2020, pp. 86–94.

[4] A. Pears, "Conveying conceptions of quality through instruction," in *Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 7–14.

[5] H. Blau and J. E. B. Moss, "FrenchPress gives students automated feedback on Java program flaws," in *ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2015, pp. 15–20.

[6] H. Keuning, B. Heeren, and J. Jeuring, "A tutoring system to learn code refactoring," in *52nd ACM Technical Symposium on Computer Science Education*. ACM, 2021, pp. 562–568.

[7] L. C. Ureel II and C. Wallace, "Automated critique of early programming antipatterns," in *50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 738–744.
[8] R. R. Choudhury, H. Yin, and A. Fox, "Scale-driven automatic hint generation for coding style," in *International Conference on Intelligent Tutoring Systems*. Springer, 2016, pp. 122–132.
[9] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
[10] O. Karnalim and Simon, "Disguising code to help students understand code similarity," in *20th Koli Calling International Conference on Computing Education Research*. ACM, 2020.
[11] W. B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*. Addison-Wesley, 2010.
[12] O. Karnalim and R. Mandala, "Java archives search engine using byte code as information source," in *International Conference on Data and Software Engineering*. IEEE, 2014, pp. 1–6.
[13] O. Karnalim and Simon, "Common code segment selection: semi-automated approach and evaluation," in *52nd ACM Technical Symposium on Computer Science Education*. ACM, 2021, pp. 335–341.
[14] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE Transactions on Knowledge and Data Engineering*, 2020.