Giving Automated Feedback About Student Code Identifiers: a Method Based on the Description of Programming Problem

Marcos Nascimento[1], **Eliane Araújo**[1], **Dalton Serey**[1], **Jorge Figueiredo**[1]

[1] Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG) – Campina Grande, PB – Brasil.

{marcosantonio@copin,{eliane,dalton,abrantes}@computacao}.ufcg.edu.br

Abstract. *Providing timely feedback on identifier naming to novice programmers can help them to improve their program readability. However, due to the growth in the number of students learning to program nowadays, giving manual feedback on identifier quality become prohibitive. In this paper, we propose a method to automatically give this feedback which is correct 75.0% of the time in contrast to the instructors' assessment. We found that 51.7% of the students who received automated feedback showed their program identifier quality improvement by picking better names. It means that we can help students to improve identifier naming and consequently, their program readability from early coding experiences.*

1. Introduction

In computer programming, identifier naming is the process of choosing names to denote code identifiers, which are constructions used to refer to, for example, variables and methods. Consequently, giving feedback on identifier quality assessment to novice programmers can help them to improve a fundamental tenet of software quality: their program readability [Butler 2009]. However, it can be tedious, error-prone and inefficient for instructors to provide this feedback manually and in a timely manner. It can also become strictly prohibitive considering the growth in the number of students learning to program nowadays in programming courses [Wilcox 2015]. In this sense, generation automated feedback on identifier quality assessment is essential to aid instructors in giving personalized one-to-one feedback.

In this paper, we propose an innovative method to generate automated feedback. When answering the programming problem, we use the description provided by the instructor to assess the appropriateness of code identifiers chosen by students. The rationale behind this idea is simple: the description includes words which can be used to construct appropriate code identifiers so that they communicate fundamental concepts of the programming problem to the reader. In this sense, the more the code expresses those concepts, the easier to read, understand, and maintain. Identifier naming based on software specification, as we propose, is also a programming style recommended in software engineering literature studies intended to improve identifier quality [De Lucia et al. 2011, Lawrie et al. 2006, Evans and Szpoton 2015].

To provide automated feedback, we go through three steps. First, we use the description of the programming problem to create the terms we use as the reference to assess the appropriateness of code identifiers: the reference vocabulary. Second, we find all of the student's code identifiers constructed without the presence of using the terms

DOI: 10.5753/cbie.sbie.2019.537

of the reference vocabulary to obtain names considered to be inappropriate. After that, we generate and provide feedback messages to the student, advising about names that could be renamed to improve software quality. We evaluated the aforementioned method conducting an in situ experiment in an introductory programming course to evaluate the effectiveness of providing automated feedback in the students' final code. We developed *IQCheck* (Identifier Quality Check) as a proof-of-concept tool that instantiates our proposal. The experiment results suggest that *IQCheck* feedback messages helps students to improve their program identifier quality. We witnessed that the students who used *IQCheck* tended to improve the appropriateness of their code identifiers, with 51.7% of the students showing better names after receiving feedback messages.

2. Related Work

There are many studies in the area of computer science education that evaluate tools to provide feedback on programming style. The works presented in these studies are related to ours regarding the intent to provide feedback on identifier quality assessment. ASSYST [Jackson and Usher 1997], CheckStyle [CheckStyle 2001], PMD [PMD 2002], STYLE [Rees 1982], Style++ [Ala-Mutka et al. 2004], among others, are tools similar to each other in the sense that they are based on the identifier name length, in characters, to provide feedback on inappropriate code identifiers. Our method differs from these tools in its methodology in the sense that to provide this feedback, it is based on the examination of student-written lexicons in code identifier constructions. Glassman et al.'s [Glassman et al. 2015] work is the closest to ours with the intent to provide feedback on inappropriate code identifiers examining their constructions. But Glassman et al.'s approach asks instructors to assess, through a user interface, the appropriateness of student-chosen identifiers based on the values those variables can take during the program execution. Our work is similar to the work presented in their study but differs in its methodology in the sense that our work aims to generate automated feedback using the description of the programming problem, without asking instructors to perform the assessment.

3. Automated Feedback Generation Method

To provide automated feedback, we go through three steps. First, we apply normalization on programming problem description to extract terms to compose a reference vocabulary. The idea is to use this reference to find appropriate code identifiers' names to student's programs. In other words, the words which could be chosen by a student to denote their code identifiers when answering the programming problem. The normalization process aims to create appropriate and relevant terms which could contribute to program readability improvement and comprehension in a bottom-up manner as the description may contain words with no relevant content or accented letters. As the first step of the normalization, we apply tokenization to the description in order to extract and obtain the lexicons denoting words. In sequence, we apply the transformation of words with capitalized and accented letters into others with non-capitalized and non-accented letters, respectively. After that, we apply the detection and removal of stop-words, which are words with little lexical content (e. g. prepositions, articles, numbers, and punctuation marks). Stop-words have no relevant or meaningful content to be written as identifiers within the code as they are unrelated to the programming problem concepts. Besides stop-words, we apply the

detection and removal of special characters, words formed by letters repeated, followed by numbers, and having a length less than two [Baeza-Yates and Ribeiro-Neto 1999]. Finally, we apply the stemming of resulting words converting plurals into singulars, transforming conjugated verb forms into infinitives, and removing suffixes from adjectives. We apply the stemming to these words as they may include plural words, conjugated verbs, and adjectives, which are inflected words to their stem. For example, "counter" and "counting" are two words inflected from the same stem, thus they would be transformed into the unique stem "count" without their suffixes "er" and "ing".

Second, we use the terms of the reference vocabulary to assess the appropriateness of student-written code identifiers when implementing their program for the original description. These are the steps of the process: Firstly, we apply the parsing of code to extract and obtain the names denoting student-chosen identifiers. In sequence, we remove duplicate names to apply the detection, division, stop-word removal of names composed of two or more terms, used to construct the identifier, separated by using the underscore or camel case separators. Secondly, we apply the stemming to resulting separated terms, or names, to obtain stems from inflected terms. Then, we find all of the identifiers constructed without the presence of using at least one term of the reference vocabulary to obtain names considered to be inappropriate. Finally, we generate and present warning messages to the student, advising which it could rename those names by better ones referencing the description of the programming problem.

4. Descriptive Case Study

In this section, we describe a study performed to investigate and demonstrate we can use the description of a programming problem for helping students to choose appropriate code identifiers.

4.1. Methodology and Data Collection

In this study, we conjectured whether we can use the words obtained from the description of a programming problem as the reference vocabulary for finding appropriate code identifiers. Considering this assumption, we examined whether using this vocabulary would allow us to find appropriate code identifiers in contrast to the way instructors do. From this point on, we will refer to the proposed method of finding appropriate code identifiers in the student program, through checking the presence of using the terms of the reference vocabulary, as "automated quality assessment". Additionally, the human assessments are those performed by the instructors of a programming course. Thus, the research question that drove our study was: **RQ1)** Does automated quality assessment, to some extent, resembles the human assessment in judging appropriate code identifiers? We evaluated the judgment of appropriate code identifiers obtained from the automated quality assessment in contrast to the study baseline using two traditional information retrieval metrics: *Precision* and *Recall*. *Precision* measures, in the context of our study, the fraction of the "number of identifiers classified as positive by the automated quality assessment, that are true identifiers according to the study baseline" by the "total number of identifiers classified as positive by the automated quality assessment". *Precision* computes the correctness of the automated quality assessment in identifying true identifiers. *Recall*, on the other hand, measures the fraction of the "number of identifiers classified as positive by the automated quality assessment, that are true identifiers according to the study baseline" by

the "total number of true identifiers according to the study baseline". *Recall* computes the completeness of the automated quality assessment when identifying true identifiers. The evaluation of these aspects, correctness and completeness, will shed light on our research question. These are the two concrete hypotheses we have formulated to evaluate the proposed technique: ($H1_1$) The automated quality assessment has acceptable correctness, considering its computed *precision* value; ($H2_1$) The automated quality assessment has acceptable completeness, considering its computed *recall* value.

To reject or accept these hypotheses, we considered the opinion of instructors took part in the study on how applicable are the levels of correctness and completeness of the proposed technique in a programming course. The null hypothesis corresponding to each alternative hypothesis listed above is that the values of the computed metrics could not be considered adequate to reinforce correctness and completeness, respectively. We assign the labels $H1_0$ and $H2_0$ to these hypotheses. In this sense, *precision* and *recall* are considered dependent variables in this study. The data corpus used in this study is composed of 125 identifiers from 58 functionally correct programs from an introductory programming course at our university in the Fall 2017. These programs were implemented in answering to a description written in Portuguese of a programming problem, asking the student to implement a program to detect and inform the player who succeeds in placing "x" or "o" in a horizontal, vertical, or diagonal row in an nxn grid recalling tic-tac-toe. The data collection was done by using an automated test-based assessment system developed in-house and tailored for the introductory programming course.

Initially, we invited five experienced instructors, from the same programming course, to assess the quality of code identifiers. We conducted a survey questionnaire asking, for each identifier, whether "(it) contributes to the program readability". To capture the instructors' opinions, this Likert-scale question, ranged from 1-5 score: "strongly disagree (1); disagree (2); neutral (3); agree (4);" and "strongly agree (5)". In our study design, *score* corresponds to the independent variable. We used Cronbach's *alpha* to measure and assess the inter-rater agreement level, or in other words, the internal consistency degree of the *scores* assigned by the instructors. The resulting *alpha* value (Cronbach's *alpha* of 0.854) was greater than 0.7 revealing that they have a high degree of agreement and that the quality of identifiers was rated similarly across the instructors [López et al. 2015]. As each identifier was evaluated by five instructors, we used the median, a location summary statistic measure, to create an average composite *score* without skewing it by any extremely large or small *scores*. After that, we classified all of the identifiers that obtained a *score* greater than the neutral value as "true" and all other values as "false". This gold-standard of "true" or "false" to identify appropriate code identifiers is our study baseline, it is used as an oracle of instructor-based identifier naming quality. In sequence, we used the automated quality assessment to assess the same code identifiers, finding appropriate and inappropriate identifiers. After that, we classified appropriate and inappropriate identifiers as "positive" and "negative", respectively, to contrast the judgment of appropriate code identifiers obtained from the automated quality assessment with the study baseline.

The illustration presented in Figure 1 shows an example of how the described methodology was applied. The first oval represents six identifiers, obtained from the data corpus used in this study, which are constructions used to refer to variables. The terms "cont_x", "cont_o", "num_bolas" and "valid", on the sequence of the automated

assessment arrow, are classified as "positive", meaning that they are appropriate identifiers according to automated assessment. These identifiers are considered appropriate as they have the presence of using terms such as "cont", "num", "bola" and "valid", which are part of the Portuguese-written description of the programming problem. The terms "cont_x", "cont_o", "i" and "num_bolas", on the sequence of the instructors' assessment arrow, are classified as "true", meaning that they are appropriate identifiers according to human assessment. Summarizing the comparative analysis of both assessment methods, the identifiers classified as "positive" and "true" are contrasted on the sequence of the intersection arrow. The terms "cont_x", "cont_o" and "num_bolas" are "true positive" (*tp*) identifiers, meaning that they are appropriate identifiers according to both assessment methods. The term "i" is a "false negative" (*fn*) identifier, meaning that even though it is classified as "true" by the instructors, it is classified as "negative" by the proposed automated assessment method. Conversely, the term "valid" is a "false positive" (*fp*) identifier, although it is classified as "positive" by the automated assessment, it is not classified as "true" by the instructors. Finally, the term "p" is a "true negative" (*tn*) identifier, meaning that it is a inappropriate identifier according to both assessment methods.

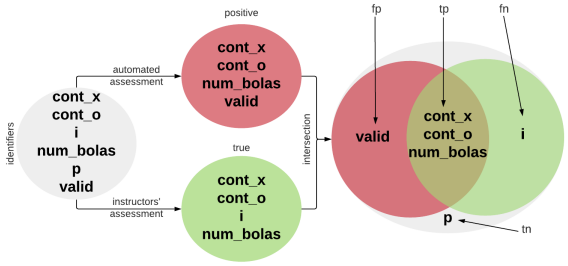

Figure 1. Contrasting instructors' manual assessment with the automated quality assessment

4.2. Results and Discussion

As a result of the comparison, we calculated the number of identifiers considered to be true positives (*tp*=57), false positives (*fp*=19), false negatives (*fn*=12) and true negatives (*tn*=37). Based on this data, we calculated the value of *precision* ($p = \frac{tp}{tp+fp} = 75.0\%$) and *recall* ($r = \frac{tp}{tp+fn} = 82.6\%$). In order to gather statistical evidence, we studied these data estimating *precision* and *recall* values. Initially, we used the ordinary nonparametric Bootstrap method to recalculate *precision* and *recall* values 2000 times by resampling identifiers classified as *tp*, *fp*, *fn*, and *tn* from the original set of identifiers, with replacement. From these values, we computed the Bootstrap confidence intervals of these metrics using the bias-corrected and accelerated method (*bca*). From the confidence interval analysis, we obtained a *precision* value of (*bca* = [63.7%; 83.5%], 95.0% confidence level) and a *recall* value of (*bca* = [71.8%; 90.4%], 95.0% confidence level).

From these values, we can observe that in our proposed automated assessment method, completeness is greater than correctness as the estimated *recall* value is relatively greater than the estimated *precision* value. Moreover, we emphasize that the values of correctness and completeness are relatively close to totality (greater than 63.0%), which means that this is a promising method for finding appropriate code identifiers. Considering the correctness aspect, we can state with adequate statistical significance that the

proposed automated assessment can correctly find most of the appropriate identifiers classified by the human assessment. Additionally, considering the completeness aspect, we can state that the proposed method can correctly find most of the appropriate identifiers in contrast to the human assessment. For these data, this means that there is evidence that the automated quality assessment, to some extent, resembles the human assessment in judging appropriate code identifiers, considering both the correctness and completeness aspects. In addition, according to instructors took part in the study, the proposed method has acceptable levels of correctness and completeness. In consequence, we reject the $H1$ and $H2$ null hypotheses in favor of the alternatives. We manually examined identifiers considered to be *fn* and *fp* to better understand why the judgment of the automated quality assessment differed from the human assessment. The first differed from the latter, showing *fn* identifiers, in assessing the appropriateness of identifiers using acronyms, abbreviations, synonyms, English terms, and single letters. On the other hand, the first differed from the latter, showing *fp* identifiers, in assessing identifiers with misleading acronyms, two or more vague words, and inappropriate naming convention.

5. Experiment

In this section, we describe an experiment conducted to investigate and demonstrate that we can help students to improve the appropriateness of their code identifiers giving automated feedback. We report on the experience of using *IQCheck* as the proof-of-concept tool that instantiates the method detailed in Section 3 to provide feedback using Portuguese-written descriptions of programming problems.

5.1. Methodology and Data Collection

In this study, we conjectured whether we can use *IQCheck* to provide feedback messages for helping students to improve the appropriateness of their code identifiers. The rationale behind this proposal is that *IQCheck* feedback messages can advise students on which their code identifiers could be renamed picking more appropriate names. Thus, our experiment was carried out to answer the following research question: **RQ2)** Does *IQCheck* feedback messages help students to improve the appropriateness of their code identifiers? We evaluated how effective are *IQCheck* feedback messages at helping students for identifier appropriateness improvement using two metrics: *rai* and Δ_{rai}. For a given student program, *rai* (the ratio of appropriate identifiers) measures the fraction of the "number of identifiers classified as appropriate by *IQCheck*" by the "total number of identifiers." It computes the ratio of identifiers classified as appropriate within the student code. Additionally, for a given pair of student programs, Δ_{rai} measures the difference between the calculated *rai* metric value of the last and first correct functionally program. It computes the difference ratio of identifiers classified as appropriate between the last and first correct program. If Δ_{rai} value is greater than zero, then it means that the number of appropriate identifiers within the last correct program is higher than in the first one, meaning that the student managed to work to deliver more appropriate identifiers.

Initially, we invited 58 students, randomly divided into control and experimental groups, to develop programs for descriptions of programming problems, thoroughly test them, and make sure they were functionally correct. The motivation behind this is that it did not make sense to over-work the student with identifier appropriateness improvement

before they had made their program effectively work, recalling the test-driven development mantra Red Bar/Green Bar/Refactor. In sequence, we invited the students from both groups to improve on the appropriateness of their code identifiers choosing names based on the given programming problem description. We then evaluated the effectiveness of providing *IQCheck* feedback messages to the students of the experimental group in contrast to the students of the control group. The students taking part in this study were enrolled in the same course mentioned in Subsection 4.1 in the Spring 2018, most of them were male (86.2%), and their ages were between 17 and 46. The data corpus used in this study is composed of 231 functionally correct programs implemented by these students in the middle of July 2018. These programs were tested and collected using the automated test-based assessment system used in the course. Those programs were included in our data corpus as the students consented to take part in this study and filled out appropriate forms approved by the committee of ethics at our university.

To provide automated feedback generation, as the first step, we instrumented the automated test-based assessment system used in the course and plugged into it *IQCheck*. In sequence, we chose descriptions of programming problems, which are traditionally used by the instructors in the course, and used *IQCheck* to create files having the reference vocabulary of each one of those descriptions. After that, we sent those files to the working directory of the students of the experimental group so that *IQCheck* could generate feedback messages after their request. Whenever the student requests *IQCheck* feedback messages, *IQCheck* uses a Python module, Abstract Syntax Trees (AST) [AST 1990], to parse the code and an implemented algorithm to extract the names denoting student-written identifiers. To create the file with the reference vocabulary, *IQCheck* uses Python modules of Natural Language ToolKit (NLTK) [NLTK 2001] such Tokenize [Tokenize 2001] to tokenize the description, NLTK's list of Portuguese stop-words to detect and remove stop-words, and RSLP Portuguese stemmer [RSLP 2001] to stem words. In addition, *IQCheck* uses a Python built-in method to convert words with capitalized; Unidecode module [Unidecode 1990] to convert words with accented letters; and RegEx module [RE 1990] to detect and remove special characters, words formed by letters repeated, followed by numbers, and having a length less than two.

5.2. Results and Discussion

We examined the Δ_{rai} metric values computed on the last and first correct programs of the students. As a result of the comparison between the groups, we found the computed Δ_{rai} metric values of the programs of the experimental group were higher than the computed Δ_{rai} metric values of the programs of the control group. It means that the students of the experimental group delivered more appropriate code identifiers in comparison to the students of the other group. To evaluate whether the control and experimental groups have the same Δ_{rai} value distribution, we examined whether Δ_{rai} value distributions of both groups differ their median value. To estimate this difference value, we applied the ordinary nonparametric Bootstrap method to compute the Bootstrap confidence interval of the difference based on 2000 resampling using the first order normal approximation method (*ona*). From the confidence interval analysis, we obtained a difference value of (*ona* = [-48.2%; -26.7%], confidence level 95.0%), which means that the Δ_{rai} values of the experimental group are higher on average than the Δ_{rai} values of the other group.

From this difference value, we can state with adequate statistical significance that

there is a significant and relevant difference between the groups. For these data, this means that there is evidence that *IQCheck* feedback messages are useful and adequate for helping students to improve the appropriateness of their code identifiers. After receiving *IQCheck* feedback messages, 15 students (51.7%) of the experimental group were able to choose better names to denote their code identifiers. In contrast with the other group, this number corresponds to five times the number of students - 3 (10.3%) - who were able to improve their code identifiers' names without receiving *IQCheck* feedback messages. The last and first correct programs implemented by these students revealed Δ_{rai} values greater than zero. In Table 1, we show some lines of the description of a programming problem translated from the original Portuguese-written description to English. In answering to the description showed in Table 1, a student could choose names, to construct their code identifiers, based on words such as "group", "number", "team", "spectator", and "stadium". In Figure 2, we show the first version of the program implemented by a given student in answering the original description, along with the feedback messages it received, in contrast to the second version of the same program.

Table 1. Description of "Largest Fan Group" Programming Problem

In the greater city of the northeastern interior, the competition for the largest fan group of the state is competitive. To solve this issue, it is required that you implement a program that reads the number of first team's spectators in 5 stadium stands and the number of second team's spectators in same 5 stadium stands. You must sum the number of spectators from each team and print the team that brings more spectators into the stadium.

Figure 2. When the student requested feedback messages, *IQCheck* provided the warnings showed in (b) regarding the identifiers in lines 2, 3, 4, 5, 7 e 8, as they were not based on words of the original Portuguese-written description. In addition to these identifiers, *IQCheck* could have marked as inappropriate others such as "cont_torcida1", "time1", "digito", "num1", and "setor5". (c) shows the program after the student actuated on the one showed in (a), renaming the identifier name in line 2 from "a" to a more appropriate: "torcida_a".

Listing 1. (a)

```
1  #coding: utf-8
2  a = 0
3  b = 0
4  for i in range(5):
5      ent = int(...
6      a += ent
7  for k in range(5):
8      ent = int(...
9      b += ent
10 ...
```

Listing 2. (b)

```
- *a* does not appear
to be a suitable name
. You should use word
s from the programmin
g assignment descript
lion.
- *i* does not app...
- *k* does not app...
- *b* does not app...
...
```

Listing 3. (c)

```
#coding: utf-8                1
torcida_a = 0                 2
b = 0                         3
for i in range(5):            4
    ent = int(...             5
    torcida_a ...             6
for k in range(5):            7
    ent = int(...             8
    b += ent                  9
...                           10
```

6. Discussion

Although identifier quality does not affect the program behavior, giving feedback on the quality of student-chosen identifiers can help to improve novice programmer training.

When answering the programming problem, the feedback allows students to reflect and improve on the names chosen by them to denote their code identifiers. The cycle request the feedback/receive/rename identifiers allow students to learn a good programming style, as they are pushed to improve identifier naming and their program readability. Although we had encouraged students to request the feedback after they had made sure their program was correct functionally, it also can be offered to functionally incorrect programs. In this case, we believe that the feedback allows students to understand why their code is wrong through the exercise of reading and renaming their code identifiers. It is important to observe that the proposed feedback does not intend to provide an exhaustive analysis of the appropriateness of identifiers, taking into account aspects such as the role, data type or relation with other identifiers which may be synonyms in the program. For this motive, the proposed method does not intend to make the distinction between words which are more relevant to denote identifiers than others, as it does not intend to check terms of high semantic relevance of the programming problem description. We believe that the code identifier quality issues of the programs we evaluated in this study are, in significant part, of the readability nature. However, as our empirical studies were not intended to prove this assumption, it is only an anecdotal suspicion that requires further analysis.

7. Threats to Validity

Human factors threaten our study validity as we invited instructors of a programming course to assess the quality of code identifiers (threat to construct validity) in different moments (threat to internal validity). Also, we used instructor-written descriptions of programming problems for finding appropriate code identifiers' names (threat to construct validity). The two first threats are mitigated as we shared the same assessment criterion with instructors and evaluated the inter-rater agreement level, respectively. The latter threat is diminished as the descriptions used were understandable, well-written, and complete according to the instructors. Although the conclusions should not be generalized to every course (threat to external validity), the ideas and methods proposed in this study can be adapted in different contexts. In this sense, we expected that instructors from other programming courses must take caution when applying the findings of this study.

8. Conclusion

We set out to provide timely feedback on identifier quality assessment so that we could help students to choose better names. To attain this aim, we proposed a method based on the description of programming problem. We performed a study to investigate and demonstrate that we can use this description to automatically find appropriate code identifiers in contrast to the way instructors do. After that, we conducted an experiment in a programming course to examine and prove that automatic feedback effectively helps students to improve on the appropriateness of identifiers at giving advice on which names could be renamed to improve software quality. The bottom line of this paper is that we can improve novice programmer training, giving students feedback about their code identifiers to improve identifier naming and their program readability.

References

Ala-Mutka, K., Uimonen, T., and Jarvinen, H.-M. (2004). Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research*, 3(1):245–262.

AST (1990). Abstract syntax trees. https://docs.python.org/2/library/ast.html. [Online; accessed 26-January-2019].

Baeza-Yates, R. and Ribeiro-Neto, B. (1999). Modern information retrieval. addison-wesley.

Butler, S. (2009). The effect of identifier naming on source code readability and quality. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, pages 33–34, New York, NY, USA. ACM.

CheckStyle (2001). http://checkstyle.sourceforge.net/. [Online; acessed 1-June-2019].

De Lucia, A., Di Penta, M., and Oliveto, R. (2011). Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Softw. Eng.*, 37(2):205–227.

Evans, E. and Szpoton, R. (2015). *Domain-driven design*. Helion.

Glassman, E. L., Fischer, L., Scott, J., and Miller, R. C. (2015). Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 609–617, New York, NY, USA. ACM.

Jackson, D. and Usher, M. (1997). Grading student programs using assyst. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM.

Lawrie, D. J., Feild, H., and Binkley, D. (2006). Leveraged quality assessment using information retrieval techniques. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 149–158.

López, X., Valenzuela, J., Nussbaum, M., and Tsai, C.-C. (2015). Some recommendations for the reporting of quantitative studies. *Computers & Education*, 91(C):106–110.

NLTK (2001). Natural language toolkit. http://www.nltk.org/. [Online; accessed 26-January-2019].

PMD (2002). A static source code analyzer. http://pmd.sourceforge.net/. [Online; accessed 1-June-2019].

RE (1990). Regular expression. https://docs.python.org/2/library/re.html. [Online; accessed 28-March-2019].

Rees, M. J. (1982). Automatic assessment aids for pascal programs. *ACM Sigplan Notices*, 17(10):33–42.

RSLP (2001). Stemmer. https://www.nltk.org/_modules/nltk/stem/rslp.html. [Online; accessed 6-April-2019].

Tokenize (2001). Tokenizer. https://www.nltk.org/api/nltk.tokenize.html. [Online; accessed 6-April-2019].

Unidecode (1990). Ascii transliterations of unicode text. https://pypi.org/project/Unidecode/. [Online; accessed 6-April-2019].

Wilcox, C. (2015). The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, New York, NY, USA. ACM.