The Role of Source Code Vocabulary in Programming Teaching and Learning

Marcos Nascimento*, Eliane Araújo†, Dalton Serey‡, and Jorge Figueiredo§
Department of Computer and Systems
Federal University of Campina Grande
Campina Grande, PB BR
*marcosantonio@copin.ufcg.edu.br, {†eliane, ‡dalton, §abrantes}@computacao.ufcg.edu.br

***Abstract*—This full paper, categorized as Research-to-Practice, presents an investigation on the role of source code vocabulary in introductory programming learning practices. According to software engineering literature, source code vocabulary is a fundamental aspect of software quality as it contributes to software readability and eases maintenance and evolution tasks. Furthermore, some studies show that identifier naming and comments in the code tend to use terms that reflect the software problem domain specification and can be used, for instance, to localize feature. In this paper, we build upon this knowledge of software engineering and investigate how we can take advantage of it in computer science education, specially on the teaching of programming. We conducted a twofold empirical study in an introductory programming course to investigate: (1) How to give automated feedback about identifier naming, aiming to improve software readability, and (2) To what extent source code vocabulary reflects problem specification comprehension, which is crucial to effective programming problem-solving. In the first study, we found that 51.7% of the students improved their source code vocabulary, after receiving such feedback. In the second study, the results showed that students tend to manage to better comprehend the programming problem being solved when their code identifier names are connected to the related description. It is a promising indicative that we can use this information to assess the problem requirements comprehension, which is a fundamental step in the programming problem-solving cycle. The main contribution of this paper is to shed light on the source code vocabulary and promote its role in the programming teaching and learning scenario. We present some evidence that students' source code vocabulary is a rich information source about their understanding and we can use it to produce formative feedback.**

***Index Terms*—computing education; automated assessment; source code vocabulary quality.**

I. Introduction

It is well-known that source code vocabulary, which is composed by the set of terms, such as identifier names and comments, plays a fundamental role on software quality [1]. When it is well-chosen, those terms give to the reader clues about code's semantic and purposes. Software engineering literature states that an adequate source code vocabulary may ease maintenance and evolution tasks, as good identifier names are usually associated with software readability [6]. According to Martin, names are responsible for 90% of what makes a software readable and understandable [17]. Additionally, using terms of the problem domain in source code vocabulary tends to increase software readability [8] [15] [10]. Source code vocabulary terms can also be used to locate features in software code [27] [24].

In our research, we investigated how we can take advantage of these findings from software engineering research regarding source code vocabulary in the context of computer science education, especially on the teaching of programming. So, we conducted a twofold empirical study to investigate: (1) How to give automated feedback about identifier naming, aiming to improve software readability, and (2) To what extent source code vocabulary reflects problem specification comprehension, which is crucial to an effective strategy to programming problem-solving.

In the first study, we focused on identifier naming - the process of choosing adequate names to denote code identifiers referring to variables and methods. We propose an innovative approach to automatically generate feedback about the quality of code identifiers considering the programming problem specification. The rationale behind this idea is simple: the specification includes fundamental concepts of the problem and using the same descriptive terms, or its derivatives, to name code identifiers can adequately communicate the problem context semantics.

We implemented *IQCheck* (Identifier Quality Check), a proof-of-concept tool to evaluate this proposal by conducting an experiment within an introductory programming course. Besides, we investigated the effectiveness of providing automated feedback in students' code of programming problem-solving exercises. We found as a result that feedback provided by the tool, referring to identifier quality, is correct 75.0% of the time having instructors' manual assessment as a baseline. Furthermore, we found that 51.7% of the students who received automated feedback improved their program identifier names. It means that this approach can help students to improve their program's readability from early coding experiences.

In the second empirical study, we investigated the link between source code vocabulary and programming problem comprehension. We conjectured that when a student includes terms of the problem specification in her code, she not only writes a more readable code but also gives instructors hints that she knows what is the program about and what it is expected to do. In this study, we observed functionally incorrect programs. These are those programs which pass some test-cases but not

all of them. Our results suggest that the more incorrect the program the less it has terms of the programming problem description.

This paper inspects the role of the source code vocabulary in the teaching of programming and claims that it is as a rich source of information that can be used to produce effective and useful feedback to learners. It is possible to assess identifier naming and provide automated feedback when contrasting it with the programming problem specification vocabulary. Such feedback goes beyond identifier naming convention rules, as it includes semantics captured from the problem domain. Furthermore, it is also a promising indicative to assess problem requirements comprehension, which is a fundamental step in the programming problem-solving cycle.

II. Related Works

Automated Assessment Systems (AAS) are tools used in programming courses to assess student-written code since 1960 [9]. AAS can generate automated and timely feedback to provide support to students in the learning of programming. In general, AAS can employ similar features [13]. The most common feature employed by these systems is feedback generation about program's functional correctness. AAS may also provide features such as quality assessment in terms of programming style to improve the visible appearance and readability. It encompasses guidelines to format programming instructions and may provide rules referring to identifier naming, commenting, indenting, and white spaces.

Many AAS apply comparable approaches when assessing identifier naming. Typically, it checks the student-chosen names against a set of naming convention rules or programming language patterns. ASSYST [14], CheckStyle [7], PMD [20], STYLE [22], Style++ [2], among others, are tools that apply such rules to help students when choosing and naming identifiers. Typically in those tools, identifiers are evaluated by its length in characters. If it is greater than or equal to a predefined size. In this case, a feedback is provided noticing that the identifier name length is lower than expected. Also, there are other rules related to multiple-word names. It suggests the use of underscore or camel case separators. We argue that these systems' approach can go beyond identifier naming rules and take into consideration semantic aspects of programming problems.

Glassman, Fischer, Scott, and Miller's work addressed identifier naming in a different way [11]. This approach is the closest to ours, as it intends to examine identifier names by observing vocabulary used in their constructions. In their work, they ask instructors to inspect, through a user interface, student-chosen identifiers grouped by the values those variables can take during the program execution. The interface helped teachers give personalized variable name feedback on thousands of student programs from an EdX introductory programming (MOOC) Massive Open Online Course. We argue that the overhead required to use the Glassman and colleagues' proposal is higher than to use our approach. We speculate that having the instructors to inspect every new identifier name and also assess them is a big hurdle to impose.

This work goes beyond suggesting an incremental improvement on current AAS, as it pushes a step further on identifier naming rules discussed in this section. In our approach, we take into consideration semantic aspects related to the programming problem context. The connections students do between the context and their code production show how it makes sense for them. This activity is a fundamental step in the learning process. According to contextual learning theory, "learning occurs only when students process new information or knowledge in such a way that it makes sense to them in their frames of reference" [12] [5].

Our proposal, drive students to create more relevant and contextualized identifier names when developing their source code. Moreover, it can help them to discover and make meaningful connections between their programming problem comprehension and the problem being solved. We speculate that when the student's code vocabulary is in agreement with the vocabulary of the programming problem specification students are led to higher comprehension and performance. Thus, relevant concepts are internalized in their mindset, through the process of discovering, reinforcing, and relating.

III. Background

In this section, we are going to briefly describe how *IQCheck* works, a proof-of-concept tool created to evaluate our approach in an introductory programming course. It generates and provides automated feedback by using programming problem specifications as input. It is worth noting that, in this evaluation course, both instructors' and students' natural language is Portuguese. This tool checks the quality of identifier names by examining source code vocabulary and contrasting it with the terms in the problem specification. As a premise of *IQCheck*, the more source code vocabulary includes the terms of the programming problem specification - hereafter called "reference vocabulary" - the easier it is to read, understand, and maintain.

The functioning of *IQCheck* is simple. It statically analyses the code to extract identifiers, in this case, variables or user-defined function names. Then, it generates and presents warning messages about those considered to be inappropriate, which are those names unrelated to at least one word or inflected terms of the reference vocabulary. *IQCheck* was built based on the premise that it should be as silent as possible, it only provides feedback messages when discover an inadequate term, to encourage learners to produce more relevant names according to the problem specification.

IQCheck performs a normalization process on the specification of programming problems to create the problem reference vocabulary. This process aims to detach from specification text single canonical terms that could be used as code identifier names within student's programs. Firstly, *IQCheck* tokenizes the specification text to extract and obtain lexicons denoting words. In sequence, *IQCheck* applies words transformation. If

TABLE I
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.

Fig. 1. (a) shows the program implemented by a given student in answering the original Portuguese-written description. When the student requested feedback messages, *IQCheck* provided the warnings shown in (b) regarding the identifiers in lines 2, 3, 4, 5, 7, and 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 "count_spectator1", "team1", "digit", "num1", and "stand5". (c) shows the program shown in (a) after the student had received useful hints on how to choose more contextualized identifier names. (c) proves that the student renamed the identifier name in line 2 from "a" to a more adequate "spectators_a". In (a), (b), and (c), the original language was translated to English so that we could show how the student's choice in (c) connected back to the description shown in Table I after the warnings shown in (b).

Listing 1. (a)

```
1  #coding: utf-8
2  a = 0
3  b = 0
4  for i in range(5):
5      inp = int(raw_input())
6      a += inp
7  for k in range(5):
8      inp = int(raw_input())
9      b += inp
10 if a > b:
11     print "The first team br
12     ings more spectators int
13     o the stadium."
14 elif b > a:
15     print "The second team b
16     rings more spectators in
17     to the stadium."
18 else:
19     print "Tie."
```

Listing 2. (b)

```
**5 Warning(s)**

### Identifiers

- *a* does not appear to be a suitable name.
You should use words from the programming as
signment description.
- *i* does not appear to be a suitable name.
You should use words from the programming as
signment description.
- *k* does not appear to be a suitable name.
You should use words from the programming as
signment description.
- *b* does not appear to be a suitable name.
You should use words from the programming as
signment description.
- *inp* does not appear to be a suitable nam
e. You should use words from the programming
assignment description.
```

Listing 3. (c)

```
1  #coding: utf-8
2  spectators_a = 0
3  b = 0
4  for i in range(5):
5      inp = int(raw_input())
6      spectators_a += inp
7  for k in range(5):
8      inp = int(raw_input())
9      b += inp
10 if spectators_a > b:
11     print "The first team br
12     ings more spectators int
13     o the stadium."
14 elif b > spectators_a:
15     print "The second team b
16     rings more spectators in
17     to the stadium."
18 else:
19     print "Tie."
```

the word contains capitalized or accented letters it is transformed in other with those letters noncapitalized and nonaccented, respectively. After that, *IQCheck* applies 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 used to name an identifier. They are unrelated to the programming problem concepts.

Besides stop-words, *IQCheck* detects and removes special characters, words formed by repeated letters, followed by numbers, and those smaller than three characters [4]. Finally, *IQCheck* applies the stemming of words, converting plurals into singulars, transforming conjugated verb forms into infinitives, and removing suffixes from adjectives. When *IQCheck* is checking code, it considers adequate those inflected words from stems of the reference vocabulary. 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".

Internally, *IQCheck* implementation uses Abstract Syntax Trees (AST) [3] Python module to parse code and an implemented algorithm to extract its names. To create the reference vocabulary, *IQCheck* uses Python modules of Natural Language ToolKit (NLTK) [19] such as Tokenize [25] to tokenize the description, NLTK's list of Portuguese stop-words to detect and remove stop-words, and RSLP Portuguese stemmer [23] to stem words. In addition, *IQCheck* uses a Python built-in method to convert words with capitalized; Unidecode module [26] to convert words with accented letters; and RegEx module [21] to detect and remove special characters, words formed by letters repeated, followed by numbers, and having a length less than three.

In Table I, we present a typical programming problem specification (also shown in our previous paper [18]). It was translated from the original Portuguese description to English. In Figure 1, we show the first version of a student program, along with the feedback messages it received, in contrast with the second version of the same program. It would be considered adequate identifier names, to answer the programming problem shown in Table I, those based on words such as "group", "number", "team", "spectator", and "stadium".

IV. RESEARCH METHODOLOGY

To investigate the source code vocabulary potential as a rich source of information on the teaching of programming, we followed a two-pronged approach: an experiment and a case study. The first aimed at evaluating: (1) how effective is our approach in assessing students' code quality, in the context of identifier names, similarly to a human does and also (2) its efficiency in stimulating students to improve their codes. The latter is a retrospective case study intended to investigate

whether using contextualized identifier names in source code relates to a better comprehension of the programming problem.

A. Data Collection

We performed this research was in an introductory programming course of a Computer Science bachelor program. In this course, students learn how to program through laboratory activities focused on solving programming problems. These activities include coding and submitting these programs to an AAS developed in-house and tailored for this course. After submitting programs, the AAS tests them executing a set of test-cases, which are basic input and output tests, previously defined by instructors for that problem. Finally, the AAS generates and delivers to students a feedback message about the program's functional correctness. A "Red Bar", signalizes that the program is incorrect, it means that it fails at least one test-case. A "Green Bar", signalizes, otherwise, a correct program that passes all test-cases. The data corpus of this research is composed of students' programs tested and collected using this AAS.

We collected 58 correct submissions, referring to one programming problem, to perform the first part of the experiment. To execute the second part, we collected 231 functionally correct programs, referring to 4 programming problems, from 58 students. The case study data corpus was composed of 397 functionally incorrect programs, referring to four programming problems, from 87 students. Students who took part in the study were enrolled in the introductory programming course, most of them male (86.2%), and their ages ranged from 17 to 46. Furthermore, their programs were included in our data corpus under their consenting in taking part in this study. This research was submitted and approved by the ethics committee of our university. The approval is under the number (CAAE) Certificado de Apresentação para Apreciação Ética 70198817.3.0000.5182 [1] since 2017.

B. Experiment

We conjectured that there are terms in the programming problem specification that represents fundamental concepts and, due to its importance, they must also exist in a program that solves such a problem. These terms can represent entities, when they name variables or actions/activities, when they designate functions, for example. These identifier names compose the source code vocabulary. In the experiment, we evaluated this conjecture, using the proof-of-concept tool *IQCheck*. Initially, we investigated if experienced instructors' assessment of identifier quality is in line with this idea. Then, we examined the effectiveness of the *IQCheck* tool. If the feedback messages it provides can adequately advise students on which code identifiers must be renamed to choose more appropriate names. We evaluated (1) how useful are *IQCheck* feedback messages at judging appropriate code identifiers in contrast with instructors' manual assessment, as a first step, and (2) how effective are *IQCheck* feedback messages at aiding students to improve the appropriateness of identifiers names in their source code. To conduct these evaluations, we formulated the following research questions:

> **RQ1)** Does *IQCheck* feedback messages, to some extent, resemble the instructors' assessment in judging the appropriateness of code identifiers?
>
> **RQ2)** Does *IQCheck* feedback messages help students to improve the appropriateness of their code identifiers?

In answering **RQ1**, we requested experienced instructors to manually assess the appropriateness of a set of identifier names, extracted from student code submissions, aiming at program readability. These activity results were used to compose this study baseline. It represents an oracle of "true positives": good identifier names. After that, we used *IQCheck* to automatically check student code submissions, to that same problem, to evaluate the appropriateness of identifier names. As described in section III, the programming problem specification must be input to *IQCheck*, so that it can evaluate and produce feedback messages about the names. Finally, we evaluated the correctness of the judgment indicated on *IQCheck* feedback messages 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 *IQCheck*, which are true identifiers according to the study baseline" by the "total number of identifiers classified as positive by *IQCheck*". *Precision* computes the correctness of *IQCheck* in identifying true identifiers.

- *Recall*: Measures the fraction of the "number of identifiers classified as positive by *IQCheck*, which 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 *IQCheck* assessment when pointing true identifiers.

The evaluation of correctness and completeness will be the answer to **RQ1**. We have formulated two concrete hypotheses to evaluate the efficacy of *IQCheck* in assessing identifier quality:

> **H1.1** *IQCheck* achieves an acceptable level of correctness, considering its computed *precision* value;
>
> **H2.1** *IQCheck* achieves an acceptable level of completeness, considering its computed *recall* value.

To verify these hypotheses, we considered instructors' opinions about correctness and completeness of feedback messages that *IQCheck* delivers when assessing students' code. Null hypotheses **H1.0** and **H2.0**, correspond to each alternative hypothesis listed above. They claim that values of computed

[1]More information can be found at http://plataformabrasil.saude.gov.br/

metrics do not reinforce correctness and completeness, respectively. In this sense, metrics *precision* and *recall* are considered dependent variables in this study.

To create the study baseline, we first invited five experienced instructors, of this introductory programming course, to assess the quality of code identifiers by answering a survey questionnaire. It asked, whether each code identifier "contributes to the program readability". We captured instructors' answers in a Likert-scale which values ranged from 1-5 score: "strongly disagree (1); disagree (2); neutral (3); agree (4);" and "strongly agree (5)". This value corresponds to the *score* and in our study design, it is an independent variable. Second, we used Cronbach's *alpha* to measure the internal consistency degree of the *scores* assigned by the instructors to assess the inter-rater agreement level. The resulting *alpha* value was greater than 0.7 revealing that they have a high degree of agreement and that the quality of identifiers was rated similarly by them [16]. 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*. Finally, we classified as "true" all identifiers that obtained a *score* greater than the neutral value and "false" otherwise.

In order to answer **RQ2**, we randomly divided 58 students into control and experimental groups. First, we assigned them 4 programming problems to develop and test. We stimulate students to test their programs and make sure they were functionally correct. As a premise, we consider that the program needs to be working before trying to make it better. We argue that it does not make sense to over-work student with identifier appropriateness improvement before the program effectively work. We recall the test-driven development mantra *Red Bar/Green Bar/Refactor*. In sequence, we prompted students from both groups to improve the appropriateness of their code identifiers by choosing names based on the given programming problem description. We then evaluated the effectiveness of providing *IQCheck* feedback messages to experimental group students in contrast to control group students. We computed two metrics: *rai* and Δ_{rai}.

- *rai*: Measures the fraction of the "number of appropriate identifiers according to *IQCheck*" by the "total number of identifiers."

- Δ_{rai}: Measures the difference between the calculated *rai* metric value of the last (*rai*1) and first (*rai*0) program.

From the computed Δ_{rai} metric, we came up with three situations:

– $\Delta_{rai} < 0$: The number of appropriate identifiers in the last submission is lower than observed in the first submission. It means that the student was not able to improve her source code vocabulary quality; instead, the last program readability became worse than the first program.

– $\Delta_{rai} = 0$: The number of appropriate identifiers in the last submission does not differ from that observed in the first submission. It means that the student does not worsen nor improve their source code vocabulary quality.

– $\Delta_{rai} > 0$: The number of appropriate identifiers in the last submission is greater than that observed in the first submission. It means that the student was capable to improve their source code vocabulary quality after renaming their code identifiers to improve its quality.

The quantitative evaluation of Δ_{rai} metric will allow us to gather evidence to answer **RQ2**. This is the concrete hypothesis we have formulated to evaluate the effectiveness of *IQCheck* feedback messages:

> **H3.1**: Students who receive *IQCheck* feedback messages, in contrast to those who do not receive, accounts a higher Δ_{rai} value between their programs.

The null hypothesis, labeled **H3.0**, corresponds to an alternative to the hypothesis listed above. It claims that students who receive *IQCheck* feedback messages, in contrast to those who do not receive, present the same Δ_{rai} value between their programs. It means that there is no evidence to reinforce a difference between control and experimental groups. In this sense, Δ_{rai} is considered a dependent variable in this study.

C. Case Study

In this case study, we hypothesized that students who tend to adequately name their code identifiers not only write a program with a better quality, but also tend to better comprehend the programming problem. We conjecture that students who comprehend the programming problem are more likely to name code identifiers using a jargon that resembles the problem being solved. In order to test this proposal, we evaluated how useful is using *IQCheck* to assess source code vocabulary quality, aiming to suggest clues about programming problem comprehension. To conduct this evaluation, we posed the following research question:

> **RQ3)** Do source code vocabulary quality in functionally incorrect programs, to some extent, indicates programming problem comprehension?

In answering **RQ3**, we used the same AAS previously mentioned to test a set of students' programs and generate feedback on their functional correctness. We used this information to identify different levels of programming problem comprehension achieved by students based on the program's functional correctness. We focused on examining the number of test-cases that were executed on those programs and the number of test-cases that they passed. After that, we used *IQCheck* to assess the same set of programs, using as input the set of programming problem specifications those programs were written for, as described in section III. Finally, we examined the relationship between the first and latter program submission in terms of two aspects assessed in the study: the program's functional correctness and source code vocabulary quality. In addition to *rai* and Δ_{rai}, the following metrics are used in this case study: *rs* and Δ_{rs}.

- *rs*: Measures the fraction of "number of test-cases that the code passess" by the "total number of test-cases."

- Δ_{rs}: Measures the difference between the value of *rs* metric of the last (*rs*1) and first (*rs*0) program submission.

V. Results

A. Experiment

We assessed the usefulness of *IQCheck* at finding appropriate code identifiers using the *precision* and *recall* metrics. Considering both correctness and completeness aspects. We claim that *IQCheck* can correctly find most of the identifiers considered to be appropriate by the instructors in their assessment. When contrasting the automatic and human assessment methods, we found *precision* (p = 75.0%) and *recall* (r = 82.6%) values. It means that correctness and completeness are relatively close to totality (greater than 70.0%). Furthermore, completeness is greater than correctness as the *recall* value is greater than the *precision* value.

Values of *precision* and *recall* approximate to those values obtained from the confidence interval analysis of *precision* (p = [63.7%; 83.5%]) and *recall* (r = [71.8%; 90.4%]) with 95.0% confidence level. These values were obtained by applying the ordinary non-parametric bootstrap method to calculate *precision* and *recall* values 2000 times by re-sampling, with replacement, identifier names from the original set of identifiers. Those values were used as a starting point to compute the bootstrap confidence intervals of the metrics using the bias-corrected and accelerated method.

At least for these data, there is evidence that *IQCheck* evaluation expressed in its feedback messages resemble, most of the time, the instructors' assessment in judging the appropriateness of code identifiers. Even though *IQCheck* feedback messages had shown some "false positives" and "false negatives", according to instructors who took part in the study, our proposal showed acceptable levels of correctness and completeness. In consequence, we rejected the **H1.0** and **H2.0** null hypotheses in favor of its alternatives. This result encouraged us to rely on the use of *IQCheck* in assessing appropriate identifier names to students' programs and, in consequence, to generate feedback messages about terms that could be renamed.

We evaluated the effectiveness of *IQCheck* feedback messages at helping students improve their identifier names. We used Δ_{rai} metric to measure this improvement. Observing the comparison between experimental and control groups, we can claim that *IQCheck* feedback messages can effectively help students to perform better in the process of choosing and renaming their code identifier names. After receiving *IQCheck* feedback messages, most of the students (51.7%) of the experimental group had quite a pretty good job by writing more appropriate code identifiers. This result contrasts with the one observed from the control group. Without receiving *IQCheck* feedback messages, very few students (10.3%) of the control group were able to improve their code identifier names. We argue that these students were able to choose better names to denote their code identifiers, as the last and first programs written by them revealed values of the metric Δ_{rai} greater than zero.

We came up with this result after observing that the comparison between the groups based on the confidence interval analysis with a 95.0% confidence level revealed a difference value of [-48.2%; -26.7%]. In practice, it means that Δ_{rai} values observed in programs of the experimental group were higher on average than those observed in programs of the control group. This result was observed after evaluating whether control and experimental groups had the same Δ_{rai} value distribution, by comparing whether the distributions of both groups differ in their median value. The difference value was estimated by applying the ordinary non-parametric bootstrap method to compute the bootstrap confidence interval of the difference based on 2000 re-sampling using the first order normal approximation method.

B. Case Study

We evaluated whether we can assess programming problem comprehension by observing source code vocabulary quality using Δ_{rai} and Δ_{rs} metrics. We observed a strong positive correlation between Δ_{rai} and Δ_{rs} values, which suggests that a program that has a better source code vocabulary quality, to some extent, can be more functionally correct than the contrary. According to the Spearman's rank correlation coefficient, the Δ_{rai} and Δ_{rs} metric values were associated with a Spearman's *rho* of 0.60. However, the positive magnitude of correlation implies that programs having a better source code vocabulary quality do not necessarily are more functionally correct.

We found in our data set 19 students' programs which Δ_{rai} and Δ_{rs} metrics presented values different from zero. Considering only Δ_{rai} and Δ_{rs} values greater than zero, we found that 10 (52.6%) of the students, that is, most of them, produced a better program in terms of source code vocabulary quality and functional correctness. It means that when the latter submission of the functionally incorrect program presented a better source code vocabulary quality, in contrast with the first functionally incorrect program submission, they also had passed a greater number of test-cases.

These students did a good job producing a more functionally correct program, at the same time they were improving their source code vocabulary quality. In contrast with these students, 3 students (15.8%) produced a worse program considering these two aspects. Curiously, 6 students (31.6%) produced an improved program depending on the aspect: 2 students (10.5%) produced an improved program only regarding the functional correctness, and 4 students (21.0%) produced an improved program only with respect to the source code vocabulary quality.

VI. Discussion and Limitations

We propose that good identifiers are those based on the programming problem specification. However, we recognize

that this strategy, embodied by the tool *IQCheck*, cannot identify all good identifiers in the code. Some of them may not derive from terms of the programming problem specification. In consequence, they can not be correctly identified by *IQCheck* feedback messages. In the experiment, 17% of the classified identifiers by *IQCheck* tool were "false negatives". A more careful look at feedback messages considered to be "false negative" revealed that they were directed to acronyms, abbreviations, relative synonyms, English terms, and single letters such as "i", "j", and "k". It means that, even though those identifiers' names had been classified as appropriate by the instructors, they had been classified as inappropriate by *IQCheck*. However, this limitation is not overly worrisome for many early programming courses.

It is possible to tune the tool to reduce the number of "false negatives". Instructors can include terms they consider to be adequate to broaden the reference vocabulary. Given that, it is possible to improve the ability of *IQCheck* in assessing the quality of identifier names, including if necessary instructors' conventions. Words and its related synonyms commonly used in codes, such as "iter" or "count" or "num" can be included as well. Additionally, from a different perspective from the observed in this study, our proposal can be tuned to suggest not only identifier naming but also comments that could be included in the students' codes.

Although we had encouraged students to use *IQCheck* after they had made sure their program was correct functionally, it also can be used with functionally incorrect programs. Our proposal is useful to highlight punctual defects, especially those observed when source code vocabulary quality appears to be associated with the program's functional correctness. Many defects can be introduced in the code when the student does not comprehend or know how to solve the problem. It is also possible that the student knows how to solve, but do not know how to program.

An anecdotal suspicion that requires further analysis, through empirical studies intended to prove this assumption, is that the defects observed in the study are, in significant part, related to the programming problem comprehension or solving strategy. So, we consider that it is important to investigate whether our proposal really can help students to understand the problem, by perceiving why their code is wrong, through the exercise of reading and renaming their code identifiers. Also, further analysis is required to probe the generalization to other programming courses.

According to this study findings, our proposal fits very well with many early programming courses where learners are urged to produce relatively short programs without a high degree of sophistication. In our empirical studies, we counted on a codebase that is of a similar scale to the problem description, which is well suited to novice programmers. It means that there is currently no proof that our proposal can be used to assess large codebases, which are possibly associated with relatively long programming problem specifications. For this reason, caution must be taken when applying our technique in different contexts as it does not intend to be a "one-size-fits-all" solution to every programming course.

VII. Conclusion

This paper presented an investigation of the role of source code vocabulary in the teaching and learning of programming. Introductory programming courses rely heavily on programming activities to develop students' skills. AASs are usually used to fasten feedback to students about the correctness of their programming problem activities. Besides functional correctness, the quality of the code must be assessed. This aspect is challenging to AASs and most of them do not explore qualitative feedback about the code in deep. We propose an approach to provide feedback on programs' readability, especially on identifier naming and problems' comprehension.

The programming problem description contains the problem specification and its terms contextualize it. We claim that using those terms when naming identifiers improves program readability. Besides, when students include terms from programming problem specification in their code, they need to reason about entities and activities in the problem. For this motive, they have a clearer understanding of the problem described in that programming specification.

Studies presented in this paper revealed that this approach can be used to identify improvement not only on identifier naming but also in students' problem specification comprehension. Our empirical studies disclosed that most of the good identifier names aiming at program readability, according to human experts, are those that include terms of programming problem specification. Additionally, we verified that after receiving feedback messages about what identifiers need to be renamed to improve their quality, most students were encouraged to choose better names from the programming problem specification. Moreover, this study also supports the idea that source code vocabulary can be used as indicative for examining programming problem comprehension. The results suggest that adherence of identifier names to the terms of specification in functionally incorrect programs is associated with a more functionally correct program. This study has gone some way towards enhancing our understanding that a better source code vocabulary quality not only increases the program comprehension but also the problem comprehension.

The main contribution of this work is the possibility of taking advantage of these findings during the phases of the programming problem-solving process. One promising application of our technique would be generating and giving feedback not only to functionally incorrect programs but also functionally correct ones. In the first case, students can be impelled to look back at their code, looking for improving identifier naming and their comprehension at the same time. We believe that, as an implication, they will take time to realize what they made incorrect, going to the point where they inserted defects in their code. In the latter case, students will have the opportunity of perceiving how they can perform better when writing quality code, especially in view of the program readability.

REFERENCES

[1] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 189–198. IEEE, 2009.

[2] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research*, 3(1):245–262, 2004.

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

[4] R. Baeza-Yates and B. Ribeiro-Neto. Modern information retrieval. addison-wesley. 1999.

[5] Stephen Billett. *Learning in the workplace: Strategies for effective practice*. ERIC, 2001.

[6] Simon Butler. 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, 2009. ACM.

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

[8] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Softw. Eng.*, 37(2):205–227, March 2011.

[9] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4, 2005.

[10] Eric Evans and Rafał Szpoton. *Domain-driven design*. Helion, 2015.

[11] Elena L. Glassman, Lyla Fischer, Jeremy Scott, and Robert C. Miller. 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, 2015. ACM.

[12] Peter Heering and Daniel Osewold. *Constructing scientific understanding through contextual teaching*. Frank & Timme GmbH, 2007.

[13] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93. ACM, 2010.

[14] David Jackson and Michelle Usher. Grading student programs using assyst. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM, 1997.

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

[16] Ximena López, Jorge Valenzuela, Miguel Nussbaum, and Chin-Chung Tsai. Some recommendations for the reporting of quantitative studies. *Computers & Education*, 91(C):106–110, 2015.

[17] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[18] Marcos Nascimento, Eliane Araújo, Dalton Serey, and Jorge César Abrantes de Figueiredo. Giving automated feedback about student code identifiers: a method based on the description of programming problem. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 537, 2019.

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

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

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

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

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

[24] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, page 212–224. ACM, 2007.

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

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

[27] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.