Function Names: Quantifying the Relationship Between Identifiers and Their Functionality to Improve Them

Charis Charitsis
charis@stanford.edu
Stanford University
Stanford, CA, USA

Chris Piech
piech@cs.stanford.edu
Stanford University
Stanford, CA, USA

John C. Mitchell
jcm@stanford.edu
Stanford University
Stanford, CA, USA

ABSTRACT

When students first learn to program, they often focus on *functionality*: does a program work? In an era where software volume and complexity increase exponentially, it is equally important that they learn to write programs with *style* so that they are readable and extendable. Writing quality code starts with the building blocks for any program, its functions. A carefully chosen name is vital for program *maintainability* and *manageability*. The identifier is the most portable and concise way to summarize what the function does. What makes for the right choice? And can we automatically assess the quality of function names? Using natural language processing, we were able to create a probabilistic model to evaluate their clarity. Using functionality encodings, we attempt to learn the relationship between functions in different programs to improve their names. We analyzed a total of 5,400 programs tackling five novice programming tasks submitted by over 1,000 students in CS1. We developed a software system to automate labor-intensive tasks, detect poor function names and recommend replacements. Our findings suggest that less than 2.5% of name substitutions have an adverse outcome, and in most cases, more than 50% result in an improvement.

CCS CONCEPTS

• **Software and its engineering → Student assessment**; • **Social and professional topics → CS1**.

KEYWORDS

CS1, common functionality detection, probabilistic model, function name assessment

ACM Reference Format:
Charis Charitsis, Chris Piech, and John C. Mitchell. 2022. Function Names: Quantifying the Relationship Between Identifiers and Their Functionality to Improve Them. In *Proceedings of the Ninth ACM Conference on Learning @ Scale (L@S '22), June 1–3, 2022, New York City, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3491140.3528269

L@S '22, June 1–3, 2022, New York City, NY, USA

ACM ISBN 978-1-4503-9158-0/22/06…$15.00
https://doi.org/10.1145/3491140.3528269

1 INTRODUCTION

Writing programs with *style* is an essential skill that often gets overlooked in CS1. In an era where software complexity increases exponentially [35], the technology industry looks for programmers who can write code that is *readable* and *resilient* to modifications. The new code a professional programmer writes per year is a small fraction of the production volume [21]. On most occasions, developers navigate through existing software rather than introduce new functionality. Developing the ability to write *quality* programs is a long process. The instructors take a lot of effort and time to provide helpful feedback. On the other hand, enrollment in CS1 courses keeps increasing and limits the available human resources per student.

Readable software originates in its building block, the function. A clear description of the performed task is vital to *maintainability* and *manageability*. Good comments explain the expected outcome concisely and highlight details or hidden corner cases. Thus, they save time to review and make the program error-resilient. Documentation begins with identifier selection. A comment that describes a function appears in a single place, its declaration. However, this function can be called from anywhere in the code. Ambiguous names in those calls can cause confusion, introduce bugs and interrupt the thought process. Replicating a comment in function calls entails the risk of outdated documentation. On the other hand, a name that captures with clarity the intended task makes the code readable. Therefore, a function identifier is the most concise and the most important form of documentation in the code. In our work, we address the following research question:

How can we automatically identify poor function names and improve them at scale?

Our paper proceeds as follows: We begin with a summary of past work on automated feedback in CS1 (Section 2). We collect a corpus of student code submissions for five particular programming challenges (Section 3). We then manually label a subset of functions with a score from 1 to 4 based on a human judgment of 'function name quality' and use machine learning to train a classifier to automatically label the rest of the corpus of functions with a score (Section 4.1). We analyze its performance and compare it with the base case and other machine learning algorithms that we considered. We proceed with a semantic comparison to identify common behavior among functions in different programs (Section 4.2). In particular, we automatically instrument all submitted solutions to print out the in-memory state at every function entrance and exit point (which represent the pre- and post-states, respectively). Next, we run a matching algorithm that detects functions in different programs where the pre- and post-states match. The algorithm tries to find a function with identical behavior and with a better

name score (Section 4.3). We proceed with threats to the validity of our work (Section 6), insight into issues we discovered while creating the model, offer our ideas to overcome them and share our experience from developing a system for automated feedback on function names (Section 7). Our paper concludes with a summary and final remarks (Section 8).

This work makes the following contributions:

- It introduces a model for function name assessment.
- It proposes a way to reveal functions with identical functionality in different programs.
- It demonstrates how to detect and improve poor function identifiers at scale.

2 RELATED WORK

Since the advent of MOOCs[25] and online CS education, there has been growing research on automated assessment tools [15]. Joy et al. describe a framework to assess programming assignments based on three principal components [18]: *functionality correctness*, *style* (well-documented code, clear layout, meaningful selection of variable and function identifiers, etc.), and *authenticity*.

There is extensive use of tools in CS1 to identify fraudulent submissions [24]. Plagiarism detection does not require program execution. Instead, the source code is compared to detect similarities between student submissions [28, 33].

Verifying the correctness can also be automated. It relies primarily on a comparison between runtime execution and expected output. Test-driven learning in CS1 has been studied extensively [7, 8, 16, 17, 29], and several automated grading tools have been deployed over the years. Web-CAT [9, 11], Marmoset [34], and ComTest [20] run automatic tests on student programs before submission to provide feedback. However, this approach requires prior knowledge of the assignment and human effort to generate unit tests. Artificial intelligence can help to lift this barrier. Piech et al. created linear mappings from an embedded precondition space to an embedded postcondition space and used them as features in a neural network to provide feedback at scale [26, 27].

The reality regarding qualitative assessment (i.e., evaluating the coding style and the program design) is quite different. On the one hand, its significance in CS education was well established early on [2, 22], and a large number of articles try to teach coding style principles using examples [14]. On the other hand, the number of tools and techniques designed to assist in coding style is undoubtedly limited. Breuker et al. tracked code properties to measure static quality in an educational setting [3]. Choudhury et al. analyzed similarities among code submissions to provide auto-generated syntactic hints to students and help them produce better-quality solutions [31]. Edwards et al. used static-analysis tools to identify problems in programs [10]. Although these tools are primarily designed to prevent bugs, they can also detect cases where the code does not comply with the programming language conventions. Regarding identifiers, they can detect names that violate capitalization conventions and are too short or auto-generated by an IDE tool.

Glassman et al. introduced a user interface for giving feedback on student variable names [13]. Foobaz is not an automatic assessment tool. It presents a scrollable list of normalized solutions accompanied by the variables occurring in the solution. Allamanis et al. developed a model that learns which function and class names are semantically similar by assigning them to locations in a high-dimensional continuous space [1]. It does that by analyzing the tokens but does not look into the actual behavior of the function. This idea works well for clustering. However, it does not guarantee that two names in close proximity correspond to functions that perform the same task.

Code style evaluation requires qualitative skills, and as such, it is commonly considered a human task. In this paper, we attempt to reverse roles. Instead of answering how machines can help humans provide feedback at scale, we try to answer how humans can teach machines to take this heavy burden away. While checking function names automatically is not a comprehensive evaluation of coding style, it is an effective automated process that provides meaningful style feedback to students. This paper builds on early-stage work that was restricted to minimalistic programming languages [4]. To examine the generality of the research method, it also considers code written in full-blown programming languages such as Java that are widely used in CS1 courses. Moreover, it delves into subtle concepts, addresses potential threats to the method validity, discusses alternative approaches, and provides insight into the challenges presented in this research work.

3 DATA COLLECTION

In this study, we gathered 5,400 programs from students who attended CS1 in four different course offerings. Of the submitted programs, 3,900 were for the initial assignment in Karel the Robot, a minimalistic programming language backed up by Java [30]. It uses a limited number of instructions (Table 1) that make Karel navigate in a world (grid) consisting of streets (rows) and avenues (columns). Karel can also place or remove beepers (diamond-shaped objects) in any given spot (corner) of the world. The remaining 1,500 programs were for two assignment challenges written in Java.

Table 1: Karel programming language commands and condition names

Built-in Karel commands		
move()	putBeeper()	pickBeeper()
turnLeft()	turnRight()	turnAround()
paintCorner(*color*)		
Karel condition names		
frontIsClear()	frontIsBlocked()	leftIsClear()
leftIsBlocked()	rightIsClear()	rightIsBlocked()
facingNorth()	notFacingNorth()	facingEast()
notFacingEast()	facingWest()	notFacingWest()
facingSouth()	notFacingSouth()	beepersPresent()
noBeepersPresent()	beepersInBag()	noBeepersInBag()

In the first and most trivial problem, dubbed "Collect Newspaper Karel," Karel the Robot leaves its house to collect the beeper from outside and returns home [36]. The second program, dubbed "Stone Mason Karel," asks the students to have Karel repair a set of arches where some of the stones (represented by beepers) are missing from the columns supporting the arches [36]. The third programming challenge, dubbed "Checkerboard Karel," begins with an M-by-N

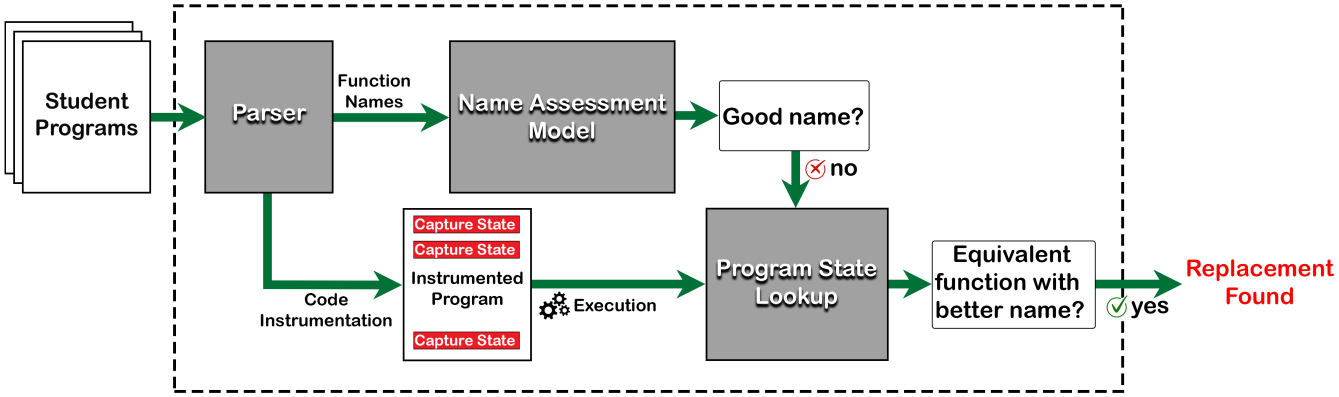

Figure 1: Method overview. Our system consists of two steps. First, it extracts the function names and feeds them to a probabilistic assessment model. Names with suboptimal scores are considered for replacement. Candidates include functions with the same functionality as the one whose name we want to replace. The selected one has the highest score among the candidates.

empty grid and asks the students to make Karel place beepers to create a checkerboard pattern [36]. The solution must work on any sized grid covering edge cases such as single-column or single-row grids, making the overall task more challenging. The first of the two Java programs dubbed "Pyramid" draws a pyramid of bricks arranged in horizontal rows centered at the bottom of the graphical window [37]. The second Java program dubbed "Yahtzee" creates a computer version of the Yahtzee game [38].

4 METHOD

Figure 1 summarizes our method. Improving function names is a two-step process: identify names with room for improvement and then find a suitable replacement for them. Thus, we need to evaluate the identifiers and recommend replacements for suboptimal scores. A candidate name needs to meet two conditions. It needs to have a higher score than the one it tries to replace, and it must perform the same task. In the following subsections, we present our assessment model (Section 4.1), explain our method to identify code with common functionality (Section 4.2), and show how to combine those two to improve function names.

4.1 Function Name Assessment

Evaluating the quality of the code is a time-consuming human task. We developed software that extracts the function names from the student submissions [5]. We ignored outliers that appeared only once and graded them manually on a scale of 1 to 4 [1]. Examples are provided in Table 2. Functions follow naming conventions in most programming languages [6, 39]. In Java, they are lowercased, and for names with multiple words, the first letter of each internal word is capitalized (e.g., *run*, *runProgram*, etc.) [23]. Our collected data shows that more than 99.6% of the students follow this camel case rule. The software applies the rule to tokenize the function names and ignores stop words that add noise [32, 41].

[1] A score of 4 means that the name is clear, a score of 3 means that it is somewhat clear with an element of ambiguity, a score of 2 means that it is unclear, and a score of 1 means that it is very ambiguous or irrelevant to the performed task.

Table 2: Examples of function names for Karel problems and their scores based on how clear they describe what to expect.

Function Name	Score	What to expect
goToWall	4	Karel moves to a location in front of a wall.
moveToTopRow	4	Karel moves to the top row.
nextRow	3	When the action verb is missing in Karel, it usually implies motion (move to next row), but we don't know for sure.
goBackToStart	3	Karel moves to the beginning of the current row or column. We don't know which of the two.
turn	2	Ambiguous. Turn left, right, up, down, or around? Hard to guess.
goKarel	1	Very ambiguous.

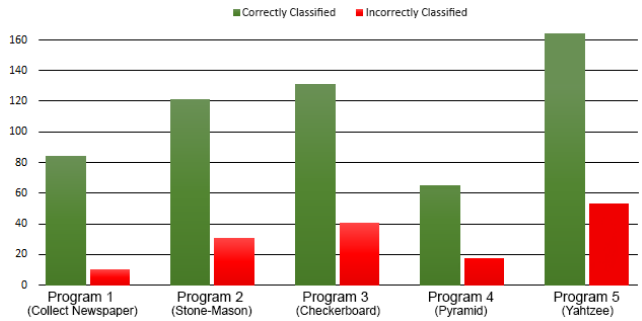

Figure 2: Classification results for different programs. The vertical axis accounts for distinct function names used by two or more students.

The remaining tokens were used to develop a probabilistic model based on the Naïve Bayes machine learning algorithm. We partitioned the data into a training and a test set (80:20 split ratio). We

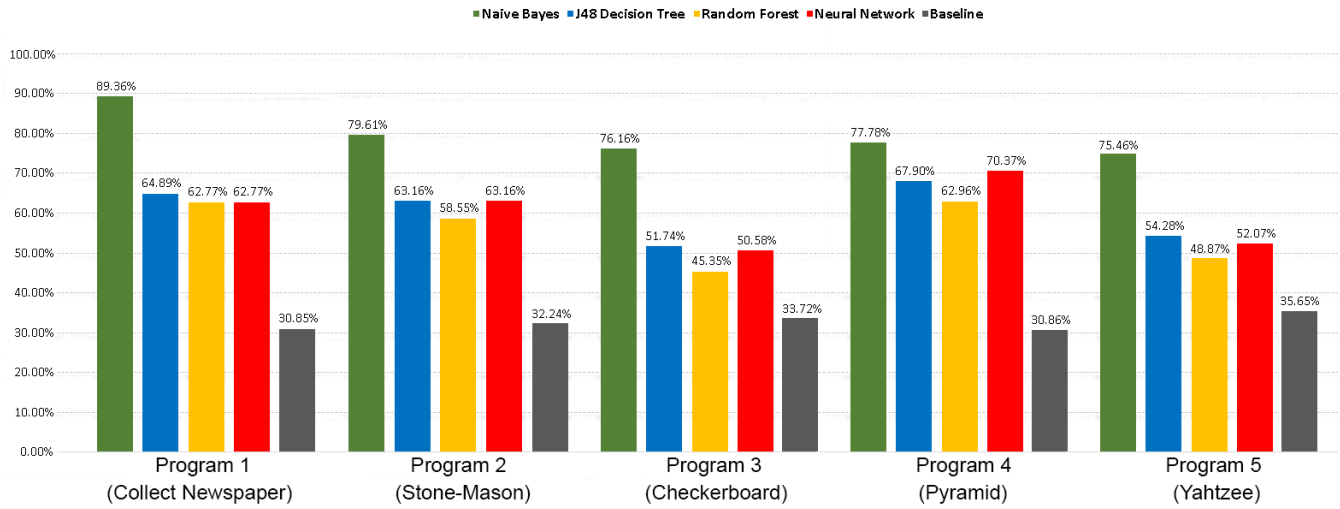

Figure 3: Correctly classified instances in our Naïve Bayes based model, other models we considered initially and the baseline.

used the scores of the function names to divide them into buckets. The prior probability that an identifier has a given score equals the bucket size for that score divided by the total number of identifiers. Approximately a third of the names got a score of 4, a third got a score of 3, and a third got a score of 1 or 2. Then, we calculated for every score the probability that a token belongs to a name of that score. After training the model, we used the test set to predict the score for unseen identifiers and compared it with the actual score. We tokenized every function name in the test data set and then multiplied the prior probability with the individual token probabilities. We calculated this product for each score (1 to 4). The one that maximizes the product is the score that our model predicts. Our software system automates the steps described in this paragraph. The only manual step is grading the function names to train the model.

In all programming challenges, the model's accuracy exceeds 75%. In the first program (*Collect Newspaper Karel*), each function completes a simple task. Thus, most students made good selections, and the number of distinct names was relatively low. The model correctly predicts 89.36% of the time. The next two Karel challenges are more complicated. Moreover, the solution does not target a specific setting. It needs to be generalized on any sized grid, adding an extra layer of complexity. As a result, the number of distinct names is higher by a factor of 1.6 in the second and 1.8 in the third challenge, and the model is correct 79.61% and 76.16% of the time, respectively. In the first Java program (*Pyramid*), the number of distinct names is low. The problem is relatively simple, and most students choose meaningful names. Nonetheless, the second Java program (*Yahtzee*) requires many helper functions, and learners seem to struggle more to select proper identifiers. The model is correct 77.78% and 75.46% of the time, respectively (Figure 2).

The first idea that comes to one's mind, which we used as the baseline, is to consider the identifier popularity as the sole assessment criterion. Intuitively, the more students choose the same function name, the higher the probability of satisfactory selection. On the other hand, students should rarely share poor names. Figure 3 shows how our model compares to the baseline that predicts the clarity score linearly from the name popularity.

In our early attempts, we considered additional attributes beyond the name popularity, such as the ratio of functions with a common name and behavior over the functions with a common name, the number of tokens in a name, etc. To find out if there is a strong relationship between them (or some of them) and clarity, we experimented with machine learning algorithms (J48 decision tree and random forest). We tried different options (bag size, iterations, pruning criteria such as confidence threshold and a minimum number of instances per leaf) to find a combination of predictors with optimum results. Similarly, we considered a neural network model, and we varied the learning rate for the backpropagation algorithm and the number of epochs to train through. Figure 3 also shows how our Naïve Bayes-based solution compares to these early approaches.

4.2 Functions With Identical Behavior

The foundation in our approach to detect code with common functionality is that a program can be considered a sequence of transitions between program states. Every computer application stores data in variables. Their contents at any given point in the execution determine the *program state* [19]. Almost every CS1 follows this paradigm, even if some unimportant variable values can be omitted. In the Karel programming language, the program state consists of Karel's world (i.e., beeper locations, etc.), the position (i.e., exact square in the grid), and the direction Karel is facing. In *Pyramid*, the number and position of the drawn bricks describe the current state. In *Yahtzee*, the scorecard and the player's turn determine the program state.

We developed software that parses the student programs and injects code to capture those states. A similar concept is used by Ernst et al. to discover invariants from execution traces [12]. Code

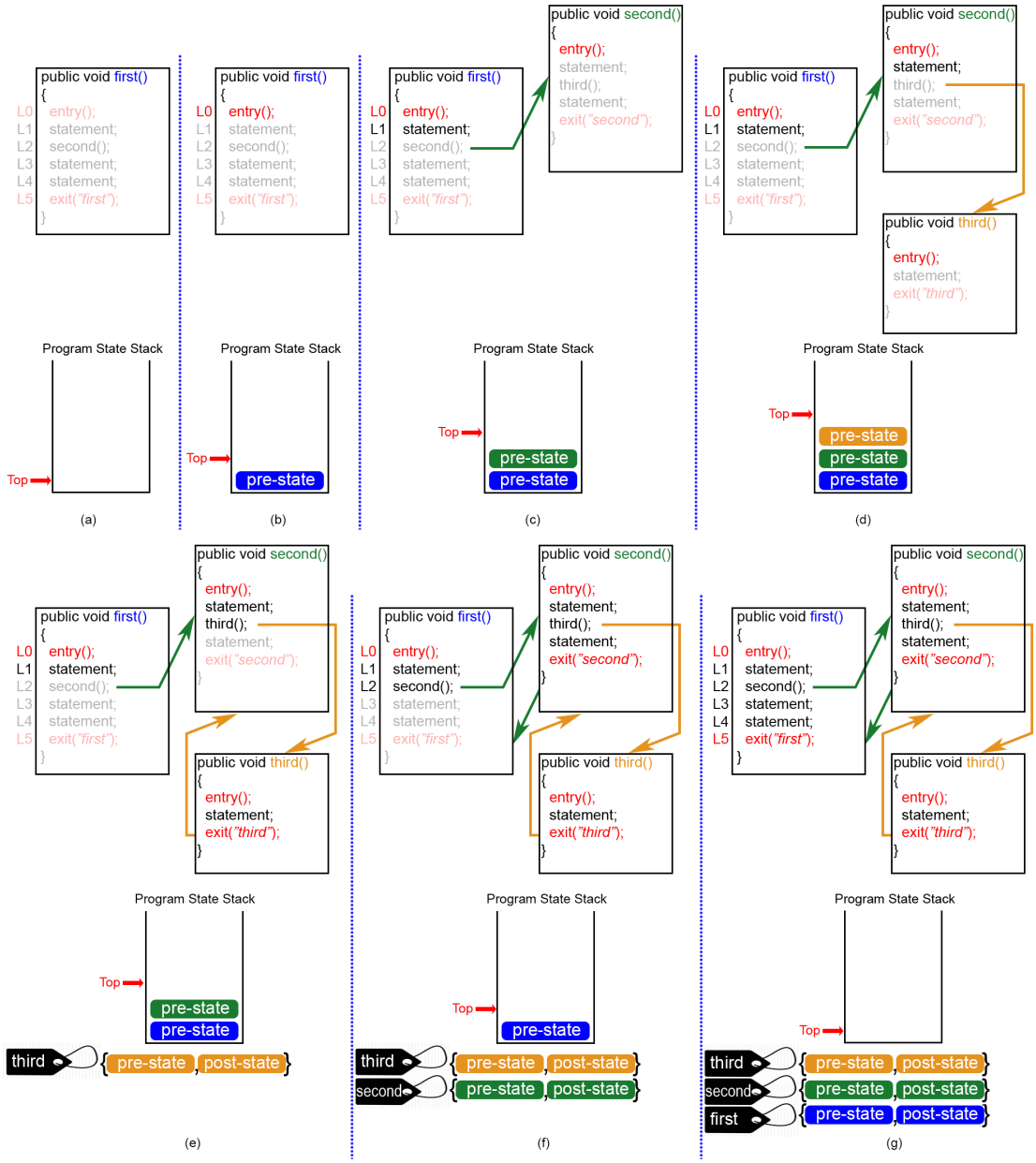

Figure 4: Code instrumentation to capture the pre/post-state pairs in nested functions. Each pair is tagged and then entered into a lookup table.

instrumentation takes place at the function level. The state is captured and pushed to a stack upon entering a function. When we exit the function, the injected code captures the state again (i.e., post-state) and pairs it with the pre-state that it pulls from the stack. The stack-oriented approach facilitates handling complicated structures such as nested functions (Figure 4). Although there is a single entry point, there can be multiple exit points (i.e., return statements), including events that disrupt the normal flow of instructions (i.e., exceptions).

The pair of pre- and post-states is inserted into a list in a lookup table with the function signature as key. The list is needed because a function can be called more than once. Figure 5 shows how to utilize the lookup table to compare functions in different programs. A matching pre/post-state pair means that the functions potentially exhibit the same behavior. However, they are **functionally equivalent** only if they are interchangeable. Thus, we must swap them and ensure that both programs produce the same output as before.

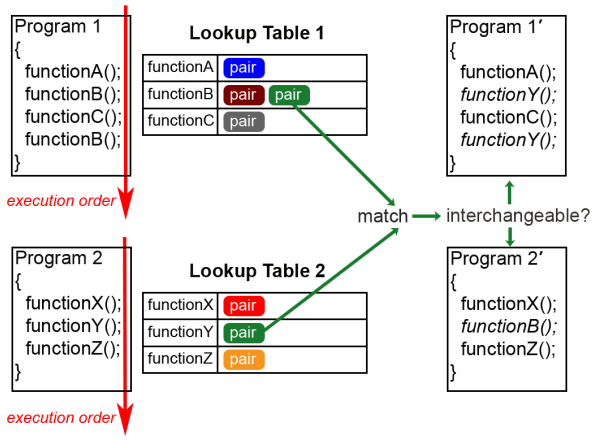

Figure 5: We create a lookup table for every program execution and then compare the stored pre/post-state pairs for potential matches. A match is a necessary but not sufficient condition for functionality equivalence. To find out if two functions are interchangeable, we must swap them and verify if both programs still produce the same output.

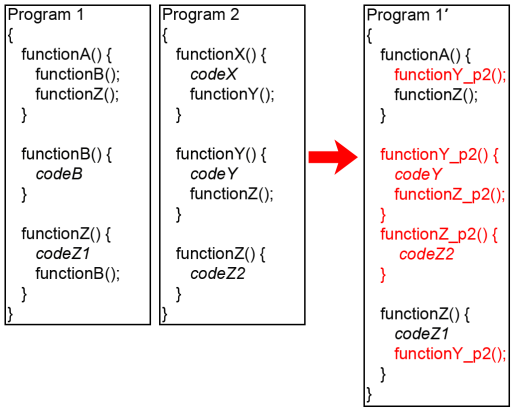

Figure 6: Source-code transformation. Pulling *functionY()* from *Program 2* to replace *functionB()* in *Program 1*. We use name mangling to resolve function name collisions.

Pulling code from one program and putting it in another can be tricky. First, one must update all function calls. Second, the pulled function may call other functions in its original program, which means that one must pull them as well. If one of them happens to have the same name as a function in the destination program, the program does not compile. To prevent collisions, we use name mangling, a mechanism used by compilers to ensure unique names for functions and variables [40]. We transform the source code by appending a unique suffix to every pulled function (Figure 6).

4.3 Function Name Replacement

After assigning scores to the function names that are extracted from the student submissions (Section 4.1), we classified the names into two groups: ones with perfect scores (i.e., 4) and ones with non-perfect scores (i.e., 1, 2, or 3) that we can further improve. To find a suitable replacement for the latter, we first need to identify candidate functions with the same behavior (Section 4.1). Finally, we can utilize our assessment model (Section 4.1) to pick the best among them. If two or more share the highest score, the most popular is selected. Table 3 provides examples of function name replacements.

Table 3: Examples of function name replacements

Function Name	Replacement
Karel (Programs 1-3)	
back	moveBackOnce
toNewspaper	moveToNewspaper
firstRow	fillFirstRow
turn	faceNextRow
returnBack	returnToStartOfRow
repair	repairColumn
Java (Programs 4 and 5)	
brick	addBrick
setRow	createRowOfBricks
sumAll	sumDice
score	updateCategoryScore
turn	playOneTurn
diceArray	rollDice

5 RESULTS

Our assessment model, which classifies the function names as perfect and non-perfect, predicts that a name can improve with high accuracy, precision, and F1 score (Table 4).

To evaluate the performance, we used a 10-fold (80:20 split ratio) and a leave-one-out cross-validation. The results account for cases where an identifier with a non-perfect predicted score was replaced by one with a higher predicted score, and their respective functions have the same behavior. Table 5 and Table 6 summarize the 10-fold and the leave-one-out cross-validation results.

The 10-fold cross-validation is more pessimistic because of the smaller training set. Larger training sets result in more accurate

Table 4: Statistical measures for the five programs

	P 1	P 2	P 3	P 4	P 5
non-perfect score (score < 4)					
Accuracy	**0.926**	**0.882**	**0.967**	**0.852**	**0.852**
Precision	**0.939**	**0.844**	**0.973**	**0.875**	**0.969**
F1	**0.898**	**0.667**	**0.863**	**0.700**	**0.854**
TPR	0.861	0.551	0.775	0.583	0.764
FPR	0.034	0.049	0.026	0.035	0.032
perfect score (score = 4)					
Accuracy	**0.926**	**0.882**	**0.967**	**0.852**	**0.852**
Precision	**0.918**	**0.817**	**0.768**	**0.846**	**0.756**
F1	**0.941**	**0.879**	**0.859**	**0.902**	**0.849**
TPR	0.966	0.951	0.974	0.965	0.968
FPR	0.139	0.049	0.245	0.417	0.236

Table 5: 10-fold cross-validation results for the five programs

	P 1	P 2	P 3	P 4	P 5
Replaced names	560	118	268	221	342
Improvement	29.46%	77.12%	89.93%	49.34%	81.57%
No impact	70.54%	22.88%	8.58%	50.66%	17.31%
Negative impact	0%	0%	1.49%	0%	1.12%

Table 6: Leave-one-out cross-validation results for the five programs

	P 1	P 2	P 3	P 4	P 5
Replaced names	635	208	485	321	602
Improvement	33.23%	83.17%	87.63%	52.86%	80.89%
No impact	66.77%	16.83%	9.90%	47.14%	17.08%
Negative impact	0%	0%	2.47%	0%	2.03%

model predictions. Moreover, there is a larger pool of candidates with common functionality to choose from when looking for replacements. For the same reason, the 10-fold has lower accuracy and F1 score (on average, 89.6% and 0.84 respectively) than the leave-one-out cross-validation (on average, 92.8% and 0.86 respectively).

Name substitutions rarely have an adverse outcome (less than 2.5% in the worst case). There is a limited number of distinct function names for the first Karel (*Collect Newspaper*) and Java (*Pyramid*) programs. Thus, the improvement was approximately 30% and 50%, respectively. However, for the other programs, the improvement reached 80% or higher.

6 THREATS TO VALIDITY

In our study, we analyzed a handful of programming challenges. *What are the threats to the generalizability of the approaches to other programming settings than those considered? Are there potential limitations?*

Comparing the Karel and Java programs (Section 5) suggests that the programming language selection can affect the results. However, a deeper analysis reveals that the problem nature has a twofold, more significant impact. First, open-ended challenges typically use more functions to decompose the main task into subtasks. Second, the problem-related vocabulary affects the word popularity and, subsequently, the individual token probabilities used in the name assessment.

The Naïve Bayes machine learning algorithm does not consider the position of the tokens in the function names. Thus, the product of probabilities is equal for two identifiers with the same tokens regardless of the order of appearance. NLP generally attempts to extract a meaning representation from raw plain text. Typically the corpus consists of full sentences that are grammatically structured. Therefore the word order matters. *Does ignoring the order pose a limitation in our case?* Function names are not grammatically structured. Thus, two names with the same tokens in different order convey almost always the same information about the performed task, and they should receive the same score. For example, even if the name *playOneTurn* reads better than *oneTurnPlay*, both reveal that the user plays one turn.

In our analysis, the scores vary between 1 and 4. It is reasonable to assume that a broader range can affect the results. We opted for a scale of 1 to 4 because of grading consistency. For example, the difference between a score of 3 and 4 is more apparent than between 9 and 10 on a scale of 1 to 10.

7 DISCUSSION

In this section, we summarize experiences and lessons learned in our investigation. *What ideas did we try? What issues did we face, and what did we do to fix them? Which tasks require more effort, and what are the future steps?*

We initially looked into the popularity of function names. Intuitively, if many students make the same selection, it must be proper and vice versa. This assumption is incorrect for two reasons. First, single-word names (usually a verb such as *build*, *check*, etc.) have a high probability because they frequently appear in student programs. However, single-word names should be accompanied by one or more words to describe the task with clarity (e.g., *buildAllRows*, *checkForWall*, etc.). Second, names with lower probability are not necessarily bad. Two identifiers with the same meaning may have quite different probabilities. For example, *createRow* has a higher probability product (Section 4.1) than *constructRow* because *create* is more a popular verb than *construct*, although they are synonyms.

Individual token probabilities make the assessment model more robust against underpredicting. Even if a token is not as popular as another with the same meaning, the remaining words can compensate for the discrepancy. However, we discovered that this logic falls short if an identifier is one word. In this case, the model overpredicts the score. To solve this issue, we penalized single-word names (i.e., point deduction from the predicted score). Another lesson was that stop words add noise, and we must ignore them. For example, *pickNewspaper* and *pickTheNewspaper* are essentially identical but produce different results because stop words skew the probabilities. Finally, we encountered a practical issue: compiling, executing, and comparing thousands of programs takes time. In-memory compilation and bytecode execution are substantially faster than operating on files in the disk. Hashing the program states increases efficiency as well.

After these adjustments, we found that the model predicts the correct score more than 75% of the time. To improve the accuracy, one can use NLP to cluster synonyms (e.g., *create*, *build*, *construct*, *make*, etc.). A downside to this approach is that many words have more than one meaning (e.g., *check* can mean *examine*, but can also mean *mark into squares*). Choosing the correct interpretation makes the model context-aware. There is a trade-off between generality and complexity one has to balance.

Most tasks are labor-intensive and practically impossible to complete without software support. We developed a system to parse the code, detect functions, tokenize identifiers, instrument the program to capture and compare its states, swap functions between programs to verify if they are interchangeable, assign clarity scores and recommend replacements for poor choices, etc. We spent more time on detecting equivalent functionality than name assessment. The only manual job is grading the names in the training set.

In the future, we plan to explore tool applications in CS1 courses. A logical next step is a built-in plugin for software development editors. The plugin will warn students about ambiguous identifiers in their code. We plan to study its impact on the adoption of good programming habits.

8 CONCLUSION

In CS1, writing code with style is a time-consuming human task that tends to be overlooked. Selecting meaningful identifiers is a fundamental skill that a programmer needs to develop early. In this paper, we presented a method for semi-automating feedback on function name quality. A proper function name has to convey with clarity the intended task. To assess the clarity of a function identifier, we introduced a probabilistic model around its tokens. To improve the names for those with non-perfect scores, we search for functions with identical behavior and better name. This paper shares the issues we discovered while developing the model and explains our ideas to overcome them. The proposed solution is context-agnostic and can be widely used. We compared the model to the baseline and other machine learning algorithms.

More than five thousand student submissions and five different programming challenges were considered in our evaluation. For our classification needs (i.e., perfect and non-perfect scores), the model exhibits consistently high accuracy, precision, and F1 score (Table 4). We developed a software system to automate labor-intensive tasks, detect poor function names and recommend replacements. Our results show that the substitutions rarely have an adverse outcome (less than 2.5% in the worst case), and they usually improve the function names.

Every CS1 course acknowledges the importance of writing clean code and tries to instill best practices. Documentation plays an essential role in software maintainability and manageability. The shortest and arguably most effective form of self-documenting code is the identifier. Yet, evaluating function names remains a human task. We hope that our work will encourage educators to incorporate some of the ideas presented in this paper in ways that fit their needs.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[2] R E. Berry and B A.E. Meekings. 1985. A Style Analysis of C Programs. *Commun. ACM* 28, 1 (Jan 1985), 80–88. https://doi.org/10.1145/2465.2469

[3] Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 13–17. https://doi.org/10.1145/1999747.1999754

[4] Charis Charitsis, Chris Piech, and John Mitchell. 2021. Assessing Function Names and Quantifying the Relationship Between Identifiers and Their Functionality to Improve Them. In *Proceedings of the Eighth ACM Conference on Learning @ Scale* (Virtual Event, Germany) *(L@S '21)*. Association for Computing Machinery, New York, NY, USA, 291–294. https://doi.org/10.1145/3430895.3460161

[5] Charis Charitsis, Chris Piech, and John Mitchell. 2022. Using NLP to Quantify Program Decomposition in CS1. In *Proceedings of the Ninth ACM Conference on Learning @ Scale* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491140.3528272

[6] Wikipedia contributors. 2022. Naming convention (programming) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Naming_convention_(programming)#Language-specific_conventions

[7] Chetan Desai, David S. Janzen, and John Clements. 2009. Implications of Integrating Test-Driven Development into CS1/CS2 Curricula. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 148–152. https://doi.org/10.1145/1508865.1508921

[8] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (Norfolk, Virginia, USA) *(SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 26–30. https://doi.org/10.1145/971300.971312

[9] Stephen H. Edwards. 2014. Work-in-Progress: Program Grading and Feedback Generation with Web-CAT. In *Proceedings of the First ACM Conference on Learning @ Scale Conference* (Atlanta, Georgia, USA) *(L@S '14)*. Association for Computing Machinery, New York, NY, USA, 215–216. https://doi.org/10.1145/2556325.2567888

[10] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/3105726.3106182

[11] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. *SIGCSE Bull.* 40, 3 (Jun 2008), 328. https://doi.org/10.1145/1597849.1384371

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/302405.302467

[13] Elena L. Glassman, Lyla Fischer, Jeremy Scott, and Robert C. Miller. 2015. Foobaz: Variable Name Feedback for Student Code at Scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, North Carolina, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 609–617. https://doi.org/10.1145/2807442.2807495

[14] Robert Green and Henry Ledgard. 2011. Coding Guidelines: Finding the Art in the Science. *Queue* 9, 11 (Nov 2011), 10–22. https://doi.org/10.1145/2063166.2063168

[15] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[16] David Janzen and Hossein Saiedian. 2008. Test-Driven Learning in Early Programming Courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 532–536. https://doi.org/10.1145/1352135.1352315

[17] David S. Janzen and Hossein Saiedian. 2006. Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) *(SIGCSE '06)*. Association for Computing Machinery, New York, NY, USA, 254–258. https://doi.org/10.1145/1121341.1121419

[18] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The Boss Online Submission and Assessment System. *J. Educ. Resour. Comput.* 5, 3 (Sep 2005), 2–es. https://doi.org/10.1145/1163405.1163407

[19] Phillip A. Laplante. 2000. *Dictionary of Computer Science Engineering and Technology*. CRC Press, Inc., USA.

[20] Vesa Lappalainen, Jonne Itkonen, Ville Isomöttönen, and Sami Kollanus. 2010. ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (Bilkent, Ankara, Turkey) *(ITiCSE '10)*. Association for Computing Machinery, New York, NY, USA, 63–67. https://doi.org/10.1145/1822090.1822110

[21] Ruchika Malhotra and Anuradha Chug. 2016. Software Maintainability: Systematic Literature Review and Current Trends. *International Journal of Software Engineering and Knowledge Engineering* 26 (10 2016), 1221–1253. https://doi.org/10.1142/S0218194016500431

[22] Susan A. Mengel and Vinay Yerramilli. 1999. A Case Study of the Static Analysis of the Quality of Novice Student Programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisiana, USA) *(SIGCSE '99)*. Association for Computing Machinery, New York, NY, USA, 78–82. https://doi.org/10.1145/299649.299689

[23] Sun Developer Network. 1999. Code Conventions for the Java Programming Language. https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

[24] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* 19, 3, Article 27 (May 2019), 37 pages. https://doi.org/10.1145/3313290

[25] Eunjung Grace Oh, Yunjeong Chang, and Seung Won Park. 2019. Design review of MOOCs: application of e-learning design principles. *J. Comput. High. Educ.* (Nov 2019). https://doi.org/10.1007/s12528-019-09243-w

[26] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep Knowledge Tracing. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 505–513. http://papers.nips.cc/paper/5654-deep-knowledge-tracing.pdf

[27] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning* (Lille, France) *(Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). JMLR, Lille, France, 1093–1102. http://proceedings.mlr.press/v37/piech15.html

[28] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8, 11 (Nov 2002), 1016–1038. http://www.jucs.org/jucs_8_11/finding_plagiarisms_among_a

[29] Viera K. Proulx. 2009. Test-Driven Design for Introductory OO Programming. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 138–142. https://doi.org/10.1145/1508865.1508919

[30] Eric Roberts. 2005. *Karel the Robot Learns Java.* https://cs.stanford.edu/people/eroberts/karel-the-robot-learns-java.pdf

[31] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-Driven Automatic Hint Generation for Coding Style. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems - Volume 9684* (Zagreb, Croatia) *(ITS 2016)*. Springer-Verlag, Berlin, Heidelberg, 122–132. https://doi.org/10.1007/978-3-319-39583-8_12

[32] Serhad Sarica and Jianxi Luo. 2020. *Stopwords in Technical Language Processing.* https://arxiv.org/pdf/2006.02633.pdf

[33] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[34] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. Association for Computing Machinery, New York, NY, USA, 13–17. https://doi.org/10.1145/1140124.1140131

[35] Steve Suh and Iulian Neamtiu. 2010. Studying Software Evolution for Taming Software Complexity. In *2010 21st Australian Software Engineering Conference* (Auckland, New Zealand). IEEE, 3–12. https://doi.org/10.1109/ASWEC.2010.26

[36] Stanford University. CS1. *Assignment 1.* https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/assn/Assignment%201.pdf

[37] Stanford University. CS1. Assignment 2. https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/handouts/Assignment%202.pdf

[38] Stanford University. CS1. Assignment 5. https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1192/handouts/36-assignment-5.pdf

[39] Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. Style Guide for Python Code. https://www.python.org/dev/peps/pep-0008//#function-and-variable-names

[40] Wikipedia contributors. 2021. Name mangling — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Name_mangling

[41] Yiming Yang. 1995. Noise Reduction in a Statistical Approach to Text Categorization. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Seattle, Washington, USA) *(SIGIR '95)*. Association for Computing Machinery, New York, NY, USA, 256–263. https://doi.org/10.1145/215206.215367