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

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

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

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

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 code with *style*. 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 3,900 programs tackling three novice programming tasks submitted by 1,300 students in CS1.

CCS Concepts
•Social and professional topics → Student assessment; CS1;

Author Keywords
CS1, common functionality detection, function names

INTRODUCTION
Writing programs with *style* is an important skill that often gets overlooked in CS1. In an era where the software complexity increases exponentially, the technology industry looks for programmers who have the ability to 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. On most occasions, developers navigate through existing software rather than introduce new functionality. Developing the ability to write *quality* programs is a long process, but it takes a lot of effort and time from the instructors to provide useful 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. 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 try to answer the following research questions:

- **RQ1: How can we automatically identify functions with the same behavior even if they belong in different programs?**
- **RQ2: How can we utilize the answer to RQ1 to detect poor function names and improve them at scale?**

We collect a corpus of student code submissions for three particular programming challenges. To answer RQ1, we present the steps for a semantic comparison to identify common behavior among functions in different programs. In particular, we automatically instrument every student's code to print out the in-memory state at every function entrance and exit point, and we run a matching algorithm that detects functions where the pre- and post-condition memory states match. To answer the RQ2, we start by manually labeling a subset of functions with a score from 1 to 4 based on a human judgment of 'function name quality' and then use machine learning to train a classifier to automatically label the rest of the corpus of functions with a score. For function names with poor scores, the algorithm tries to find a function with identical behavior and a better name score.

RELATED WORK
Since the advent of MOOCs and online CS education, there is growing research on automated assessment tools. Joy et al. describe a framework to assess programming assignments based on three principal components [6]: *functionality correctness*,

ACM ISBN 978-1-4503-8215-1/21/06.
http://dx.doi.org/10.1145/3430895.3460161

style (well-documented code, clear layout, meaningful selection of variable and function identifiers, etc.), and *authenticity*.

Identifying fraudulent submissions is the easiest of the three to automate [7]. Verifying the correctness can also be automated. Many systems run automatic tests on student programs to provide feedback [3]. However, this approach requires prior knowledge of the assignment and human effort to generate unit tests. Artificial intelligence can help to lift this barrier [8].

The reality regarding qualitative assessment is different. The number of tools and techniques designed to assist in coding style is limited in volume and capacity. Breuker et al. tracked code properties (i.e., counted lines of code, comments, number of functions, blank lines, etc.) to measure static quality in an educational setting [2]. Choudhury et al. analyzed similarities among code submissions to provide auto-generated syntactic hints to students and help them produce better-quality solutions [10]. Glassman et al. introduced a user interface for giving feedback on student variable names [5]. 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].

DATA COLLECTION

We gathered 1,300 submissions from CS1 students in four different course offerings. Each submission contains solutions to three problems. Thus, we analyzed 3,900 submitted programs for the initial assignment in Karel the Robot, a minimalistic programming language backed up by Java [9]. It uses a limited number of instructions that make Karel navigate in a world (grid) consisting of streets (rows) and avenues (columns). Besides moving vertically and horizontally in the grid, Karel can place one or more beepers (diamond-shaped objects) in any given spot (corner) of the world identified by its row and column. Karel can also remove one or more beepers that exist in a corner. The Karel programming language exposes students only to the fundamental concepts first taught in CS1. A graphical interface helps the students visualize the problem they need to solve and verify its correctness upon execution.

IDENTIFYING FUNCTIONS WITH IDENTICAL BEHAVIOR

Most systems analyze the behavior of a program to evaluate its correctness. In our study, we were interested in detecting functions with identical behavior. The basis of our idea is that every computer program stores data in variables. Their contents at any given point in the program's execution determine the *program state*. Almost every CS1 follows this paradigm. 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.

Our method consists of four steps: a) *code instrumentation*, b) *in-memory compilation*, c) *bytecode execution*, and d) *pairwise state comparison*. In every function, we add an entry command before the first statement in its body and an exit command right before it returns to capture the *program state*. A similar concept is used by Ernst et al. to discover invariants from execution traces [4]. The entry command captures the state before execution (i.e., pre-condition) and pushes it into a stack. The exit command pairs the state after execution

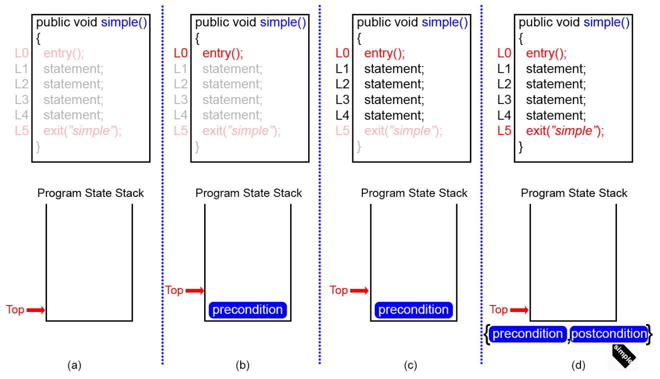


Figure 1. Function execution: a) before the *entry-commend*, b) before the first statement, c) after the last statement, d) after the *exit-command*.

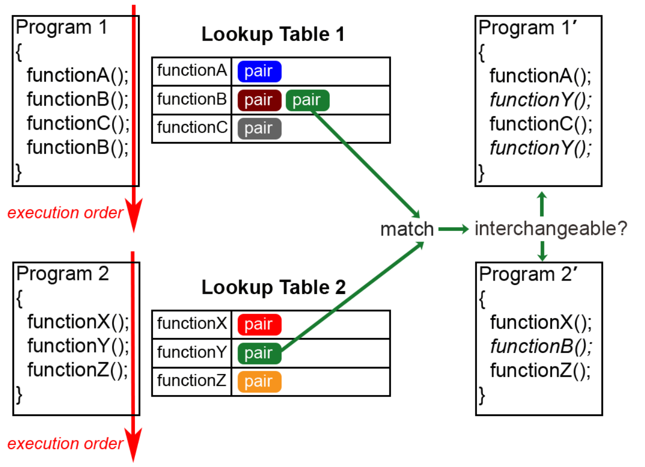
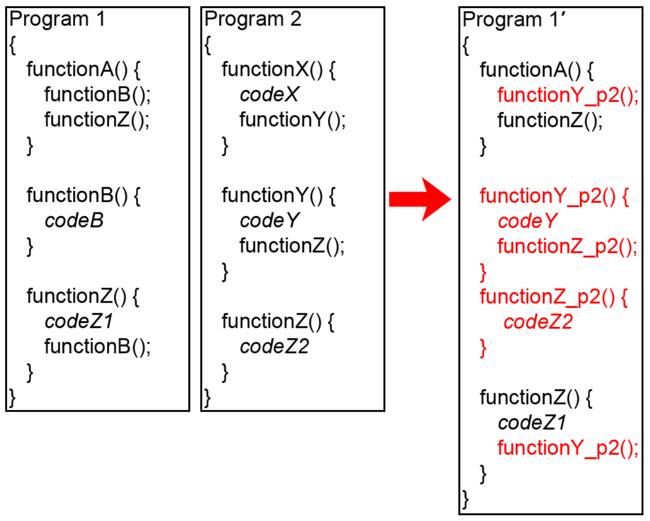


Figure 2. For every program execution, we create a lookup table and then compare the stored pre/post-condition pairs for potential matches. A match is a necessary but not sufficient condition for functionality equivalence. To find out if two functions are interchangeable, we must swap them and verify if both programs still produce the same output.

(i.e., post-condition) with the pre-condition that it pulls from the stack (Figure 1). The stack-oriented approach facilitates handling complicated structures such as nested functions. A function can return at multiple points. Thus, the code instrumentation must insert an exit command before each return point, including compile-time exceptions.

A program can call a function one or more times. Every pre/post-condition pair is inserted into a lookup table using the function name as its key. Figure 2 shows how to utilize the lookup table to compare functions in different files. We compare the lookup tables for the two programs. If we find a matching pre/post-condition pair between two functions, it means that they potentially exhibit the same behavior. To determine if they exhibit identical behavior, we must swap them and make sure that both programs produce the same output as before.

One needs to be careful when pulling code from one program and putting it to another. First, all function calls must be updated. Second, the function that is pulled may call other functions in its original program, which means that we must



```
Program 1
{
  functionA() {
    functionB();
    functionZ();
  }

  functionB() {
    codeB
  }

  functionZ() {
    codeZ1
    functionB();
  }
}
```

```
Program 2
{
  functionX() {
    codeX
    functionY();
  }

  functionY() {
    codeY
    functionZ();
  }

  functionZ() {
    codeZ2
  }
}
```

```
Program 1′
{
  functionA() {
    functionY_p2();
    functionZ();
  }

  functionY_p2() {
    codeY
    functionZ_p2();
  }
  functionZ_p2() {
    codeZ2
  }

  functionZ() {
    codeZ1
    functionY_p2();
  }
}
```

Figure 3. Source-code transformation. Pulling functionY() from program2 to replace functionB() in program1. Name mangling is used to resolve collisions between the function names.

pull them as well. If one of them happens to have the same name as a function in the destination program, it does not compile. To prevent collisions, we use name mangling. We transform the source code by appending a unique suffix to every pulled function (Figure 3).

FUNCTION NAME ASSESSMENT

We developed software that extracts the function names from the student submissions. We ignored outliers that appear only once and graded them manually on a scale of 1 to 4, where a higher value means more clarity (Table 1). Functions follow naming conventions in most programming languages. In Java, they are lowercased, and for names with multiple words, the first letter of each internal word is capitalized. We applied this rule to tokenize the names and ignored stop words that add noise.

Function Name	Score	What to expect
goToWall	4	Karel moves to a location in front of a wall.
goBackToStart	3	Karel moves to the beginning of the current row or column. We don't know which of the two, though.
nextRow	3	Karel probably moves to the next row, but we do not know the action.
turn	2	Ambiguous. Turn left, right, up, down, or around? It is hard to guess.
goKarel	1	Very ambiguous.

Table 1. Examples of function names and their scores based on how clear they describe what to expect.

The remaining tokens were used to develop a probabilistic model. The data was partitioned into a training and a test set. We 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. Then, we calculated for every score (1 to 4) 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 for programs written in Java. The only manual step is grading the function names to train the model.

In all three programming challenges, the model's accuracy exceeds 75%. In the first and most trivial program, each function completes a simple task. Thus, most students made good selections, and the number of distinct names was relatively low. The model predicts correctly 89.36% of the time. The last two challenges are more complicated. 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 (Figure 4). Figure 5 shows how our model compares to the baseline that predicts the clarity score linearly from the name popularity.

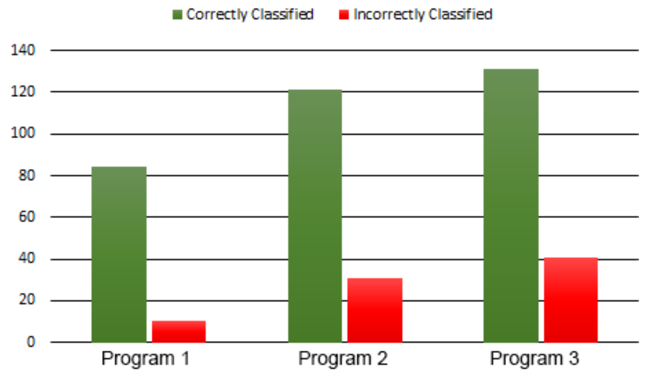


Figure 4. Number of unique function names in the student submissions for each program that are correctly/incorrectly classified. Function names used by only one student (i.e., outliers) are excluded.

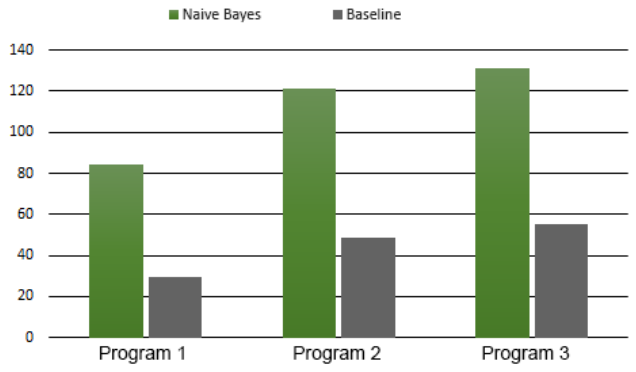


Figure 5. Number of unique function names in the student submissions for each program that are correctly classified. Function names used by only one student (i.e., outliers) are excluded.

IMPROVING FUNCTION NAMES

Improving function names is a two-step process: identify names with room for improvement and then find a suitable replacement for them. Thus, we classified them 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. A candidate replacement name has 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. Identifying code with common functionality allows to narrow down the set of candidates. We can then utilize our assessment model to pick the best among them. If two or more share the highest score, the most popular is selected. Our assessment model to classify the function names as perfect and non-perfect predicts that a name can improve with high accuracy, precision, and F1 score (Table 2).

	Program 1	Program 2	Program 3
non-perfect score (score < 4)			
Accuracy	**0.926**	**0.882**	**0.967**
Precision	**0.939**	**0.844**	**0.973**
F1	**0.898**	**0.667**	**0.863**
perfect score (score = 4)			
Accuracy	**0.926**	**0.882**	**0.967**
Precision	**0.918**	**0.817**	**0.768**
F1	**0.941**	**0.879**	**0.859**

Table 2. Statistical measures derived from the confusion matrices

To evaluate the performance, we used a 10-fold (80%-20% splitting) cross-validation. The results (Table 3) 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 4 provides examples of function name improvements.

	Program 1	Program 2	Program 3
Replaced names	560	118	268
Improvement	**29.46%**	**77.12%**	**89.93%**
No impact	70.54%	22.88%	8.58%
Negative impact	**0%**	**0%**	**1.49%**

Table 3. Results for 10-fold cross-validation

Function Name	**Replacement**
back	moveBackOnce
toNewspaper	moveToNewspaper
firstRow	fillFirstRow
turn	faceNextRow
returnBack	returnToStartOfRow
repair	repairColumn

Table 4. Examples of function name improvements

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 (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 38–49. DOI: http://dx.doi.org/10.1145/2786805.2786849

[2] 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 (ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 13–17. DOI: http://dx.doi.org/10.1145/1999747.1999754

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

[4] 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 (ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 213–224. DOI: http://dx.doi.org/10.1145/302405.302467

[5] 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 (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 609–617. DOI:http://dx.doi.org/10.1145/2807442.2807495

[6] 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. DOI: http://dx.doi.org/10.1145/1163405.1163407

[7] 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. DOI:http://dx.doi.org/10.1145/3313290

[8] 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

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

[10] 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 (ITS 2016)*. Springer-Verlag, Berlin, Heidelberg, 122–132. DOI: http://dx.doi.org/10.1007/978-3-319-39583-8_12