Using NLP to Quantify Program Decomposition in CS1

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

Decomposition is a problem-solving technique that is essential to software development. Nonetheless, it is perceived as the most challenging programming skill for learners to master [31]. Researchers have studied decomposition in introductory programming courses through guided experiments, case studies, and surveys. We believe that the rapid advancements in scientific fields such as machine learning and natural language processing (NLP) opened up opportunities for more scalable approaches.

We study the relationship between problem-related entities and functional decomposition. We use an automated system to collect 78,500 code snapshots from two CS1 programming assignments of 250 students and then apply NLP techniques to quantify the learner's ability to break down a problem into a series of smaller, more straightforward tasks. We compare different behaviors and evaluate at scale the impact of decomposition on the time it takes to deliver the solution, its complexity, and the student's performance in the assignment and the course exams. Finally, we discuss the implications of our results for teaching and future research.

CCS CONCEPTS

• **Software and its engineering → Software development methods**; • **Social and professional topics → CS1**.

KEYWORDS

problem decomposition, problem-solving approaches, introductory programming courses, software development

ACM Reference Format:
Charis Charitsis, Chris Piech, and John C. Mitchell. 2022. Using NLP to Quantify Program Decomposition in CS1. 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, 8 pages. https://doi.org/10.1145/3491140.3528272

1 INTRODUCTION

One of the fundamental human processes in software development is problem-solving. It is a cognitive process of the brain that searches or infers a solution for a given problem in the form of paths to reach expected goals [34]. Researchers in computer science education have long been interested in the relationship between problem-solving skills and programming abilities [25]. The strategies that novices adopt while solving a problem affect their performance significantly [17, 32].

Researchers and educators have studied various problem-solving approaches to software development (Section 2). The two most recognized design paradigms are top-down and bottom-up programming. The top-down development starts by implementing the most general modules and works toward those that provide specific functionality. The bottom-up development first implements the modules that provide specific functionality and then integrates them into more general modules. Research to evaluate their impact on introductory programming courses is structured around case studies, experiments, and tools [20, 23, 30, 36]. Although these scientific studies are undeniably helpful, they pose limitations. First, they use practices (i.e., interviews, surveys, etc.) that do not scale. Second, they guide students to exhibit prelabeled behaviors, enabling comparison between those behaviors. Thus, they lack generality. For example, researchers have a consensus about the connection of top-down programming with abstraction and problem decomposition, two of the essential skills in software development. In practice, there is no pure bottom-up or pure top-down design, even in CS1, where the programs are simple. A problem solver attempts several approaches on the path to the final solution [10, 14].

Problem decomposition is not binary. As a higher-layer cognitive process, it interacts with many other cognitive processes such as abstraction, decision making, inference, analysis, and synthesis. It manifests to a different extent among individuals. Similarly, there is a spectrum for the time it unveils. In some solutions, decomposition becomes apparent early on, while it is not evident until later in others. It would be immensely beneficial if we could find where a student's solution lies in the spectrum and compare it with the solutions of other learners. This motivated our study, which tries to answer two research questions:

(1) **How can we quantify the student's ability to break a problem into simpler tasks from the source code?**
(2) **How does this metric relate to the time it takes to deliver the solution, its complexity, and the student's performance?**

Our collected data consists of code snapshots taken for two CS1 programming assignments. We developed a model to apply natural language processing (NLP) techniques and learn the main entities associated with the programming challenge under investigation. We tracked how a student's program evolved with respect to these entities. Learners with an early perspective about the end goal decompose the program into functions that address the main entities sooner than learners with a narrower view of the problem. We used a model to quantify this ability and compare different solutions. We clustered the students into two groups: i) those who have a more abstract view of the problem to solve and ii) those who focus on specific subtasks of the problem first and then integrate them. Finally, we analyzed the effect of those behaviors regarding the time it takes to complete the program, its complexity, and the student performance in both the programming assignments and the course exams. The following sections elaborate on our method and our evaluation results. The primary contributions of this paper are the following:

- Application of NLP concepts to a new domain
- Conversion of a labor-intense human process to a machine task
- Gained insights and potential applications in CS education

This work is particularly pertinent to introductory programming classes with a large audience, lecture-based or online.

2 RELATED WORK

Researchers consider problem decomposition a critical skill in software development. McCracken et al. developed a five-step framework of expectations for learning from CS1 courses: 1) abstract the problem from its description, 2) generate subproblems, 3) transform subproblems into sub-solutions, 4) re-compose the sub-solutions into a working program, and 5) evaluate and iterate [22].

Keen and Mammen detail a term-long course project. Students are given clear direction on program decomposition at early milestones and progressively less direction at later milestones [13]. They compare the cyclomatic complexity [21] of final assignment submissions between students in a course with the term-long project and their cohorts in courses with stand-alone projects. Their analysis supports that long-term projects are more beneficial. Sooriamurthi discusses a learning exercise where CS1 students have to break a large programming assignment into smaller pieces and concludes that it emphasizes abstraction, decomposition, incremental and iterative development compared to simpler independent programming problems [33].

Pearce et al. implement a guided inquiry-based learning approach to teach students strategies for problem decomposition in a CS1 course [26]. They utilize a rubric to measure the student's ability to decompose problems. They examine the final projects from two offerings of the CS1 course: one with and one without explicit instruction on problem decomposition. Their findings suggest that the students in the section with the additional scaffolding were significantly more adept at breaking down the problem into subproblems.

Our method differs in two ways. First, it does not require the formation of two groups, one guided toward problem decomposition and one not [13, 17, 24, 26, 33]. Instead, we analyze submissions from the same pool of students and try to identify the ones where the presence or absence of decomposition relative to their peers is evident. We believe that this approach is generic and has more practical implications. Although surveys, think-aloud experiments, and guided-based research techniques are undeniably insightful, it takes a lot of effort to apply them at scale. One can argue that the safest way to reach objectiveness is not to rely primarily on what the students think but on their actual source code and analyze the student program structure. Doing so by hand is inefficient and time-consuming, but we believe we have the tools for a computer-assisted exploration of programming solutions nowadays.

Natural Language Processing (NLP) is widely integrated with many educational contexts such as linguistics, e-learning, evaluation systems, etc. [1, 7] However, its use in programming is limited to extracting useful information from the source code. Matskevich and Gordon present a method to generate informative comments from the source code using NLP [19]. Fowler et al. utilize NLP techniques to automatically grade plain-text answers to code reading questions. Charitsis et al. introduce a method to detect poor function names and recommend replacements [4]. We apply NLP to a new domain. We build a system that parses the student code and uses a model to detect the main problem entities, a key step in our method to determine how students decompose their programs.

3 METHOD

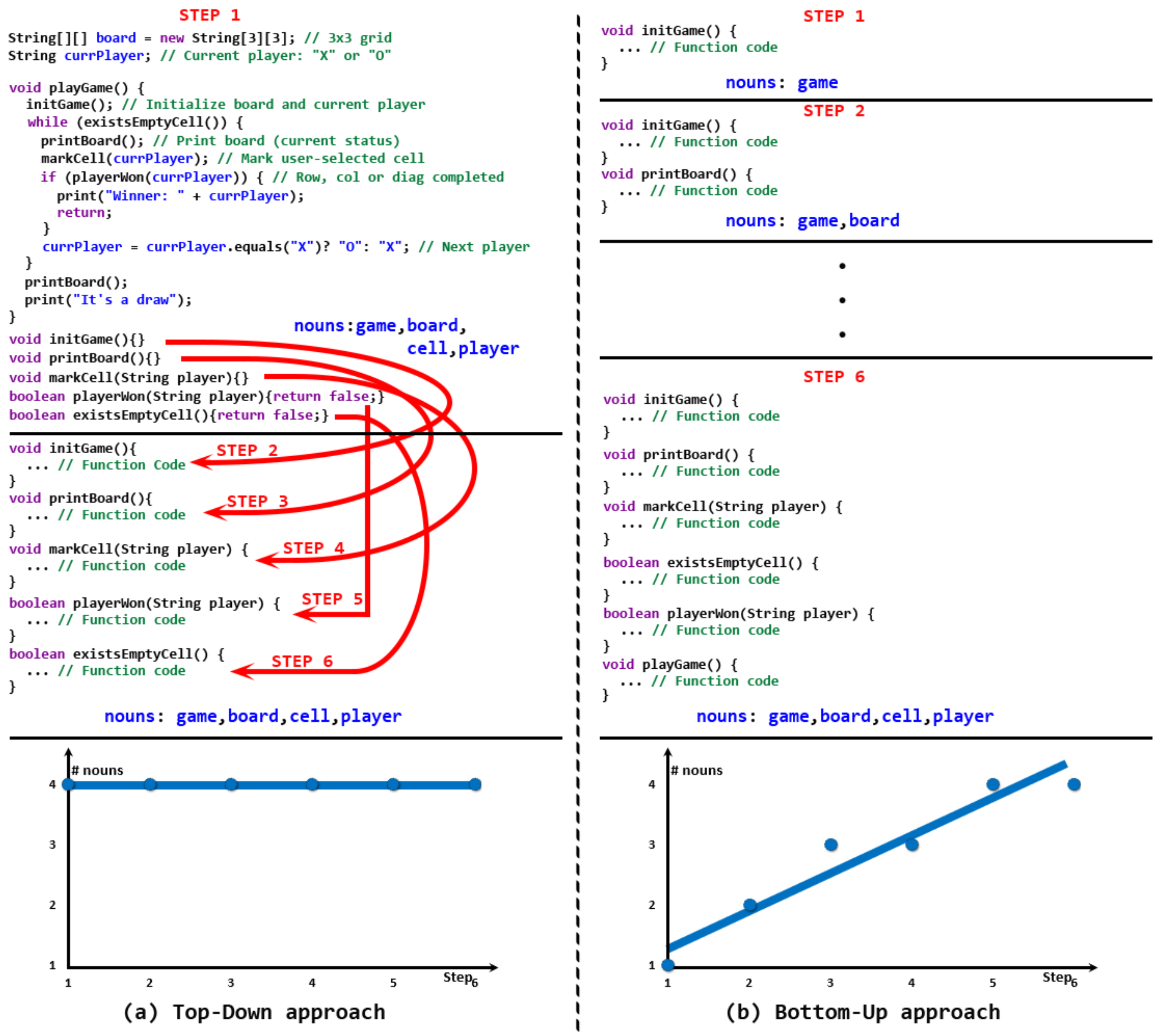
The foundation of our method is to examine how the novice learner addresses key problem tasks in the development phase. Figure 1 uses a program written in Java to demonstrate two opposite approaches. On the left side, a student uses stub functions to draft an initial solution (step 1) and then adds the implementation for each function (steps 2-6). On the right side, a second student implements tasks with specific functionality first and then integrates those functions into the final solution. The end product, which may be identical in both cases, does not reveal the software development path.

In object-oriented programming, objects (entities) are often referred to as *nouns* and actions that determine their behavior as *verbs*. Function names systematically couple these verbs to the entity associated with the action. The program on the left introduces all functions in the beginning, and therefore all nouns (i.e., *game*, *board*, *cell*, *player*) are present. The program on the right adds the functions incrementally. The student implements *initGame()* first and then moves to the subsequent function, *printBoard()*. Figure 1 shows at the bottom how nouns are introduced over time in both programs.

The example made two simplifications to highlight the basic concept of our method. First, novice programmers do not develop their programs using either a top-down or a bottom-up approach but a combination of both. Second, not all functions use nouns, and if they do, those do not always reflect on key problem tasks (i.e., the entities are not *main* entities). Helper functions can also refer to nouns unrelated to the problem and must be ignored. We will address both later in this paper. Figure 2 presents our system's architecture to quantify program decomposition from the source code (stages 1-4) and analyze its impact on program complexity and student performance (stage 5).

3.1 Data Collection

We modified the Integrated Development Environment (IDE) that CS1 students use to develop their programs to commit a source code snapshot to a local repository every time a student saves or tries to run the program. When students submit the assignment solution, they can opt-in to also submit the repository with the snapshots taken in their problem-solving journey. For this investigation, we analyzed the submissions for two assignments by 250 students, a total of 78,500 code snapshots written in the Java programming



```
String[][] board = new String[3][3]; // 3x3 grid
String currPlayer; // Current player: "X" or "O"

void playGame() {
  initGame(); // Initialize board and current player
  while (existsEmptyCell()) {
    printBoard(); // Print board (current status)
    markCell(currPlayer); // Mark user-selected cell
    if (playerWon(currPlayer)) { // Row, col or diag completed
      print("Winner: " + currPlayer);
      return;
    }
    currPlayer = currPlayer.equals("X")? "O": "X"; // Next player
  }
  printBoard();
  print("It's a draw");
}
void initGame(){}
void printBoard(){}
void markCell(String player){}
boolean playerWon(String player){return false;}
boolean existsEmptyCell(){return false;}
```

```
void initGame(){
  ... // Function Code
}
void printBoard(){
  ... // Function code
}
void markCell(String player) {
  ... // Function code
}
boolean playerWon(String player) {
  ... // Function code
}
boolean existsEmptyCell() {
  ... // Function code
}
```

```
void initGame() {
  ... // Function code
}
```

```
void initGame() {
  ... // Function code
}
void printBoard() {
  ... // Function code
}
```

```
void initGame() {
  ... // Function code
}
void printBoard() {
  ... // Function code
}
void markCell(String player) {
  ... // Function code
}
boolean existsEmptyCell() {
  ... // Function code
}
boolean playerWon(String player) {
  ... // Function code
}
void playGame() {
  ... // Function code
}
```

Figure 1: The two opposite approaches to write a program that plays Tic Tac Toe: a) The top-down starts by implementing the general function *playGame()* and works toward functions that implement specific subtasks. b) The bottom-up implements helper functions in the beginning (*steps 1-5*), and in the end, (*step 6*) calls them in function *playGame()*. We plot the number of nouns that appear in the function names in every step in each approach.

language. The assignments are structured such that the deliverable tasks are well-defined [2, 3]. The students are given explicit directions to clarify the expected functionality and ensure that the problem is well understood. Also, the description order probably aids the inexperienced student that is often unsure of how to begin. Nevertheless, the program decomposition is left to the student. The first stage completes upon extraction of the snapshots and their timestamps from the local repository.

3.2 Parser

A parser processes every snapshot retrieved from the local repository. The role of the parser is dual: 1) it creates a list with the function names in the program, and 2) it captures metrics related to the program's complexity to evaluate decomposition (Section 4). There are many code-based software complexity measures [11, 15, 16, 21, 35]. Like Keen and Mammen [13], we used McCabe's cyclomatic complexity metric and tracked the software lines of code (SLOC). Moreover, we identified a potential inconsistency in

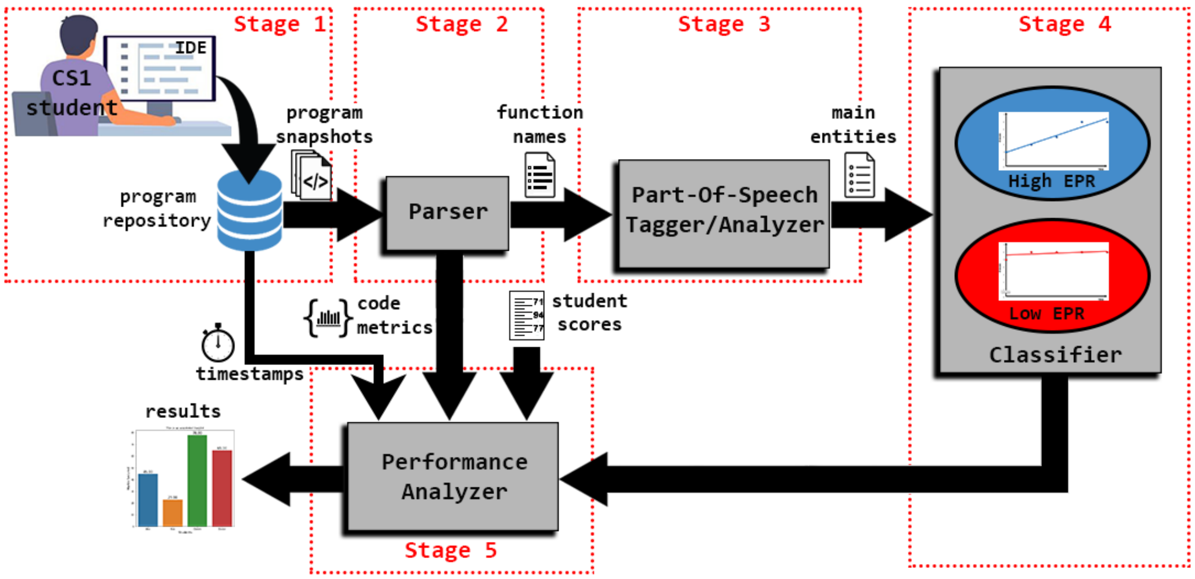
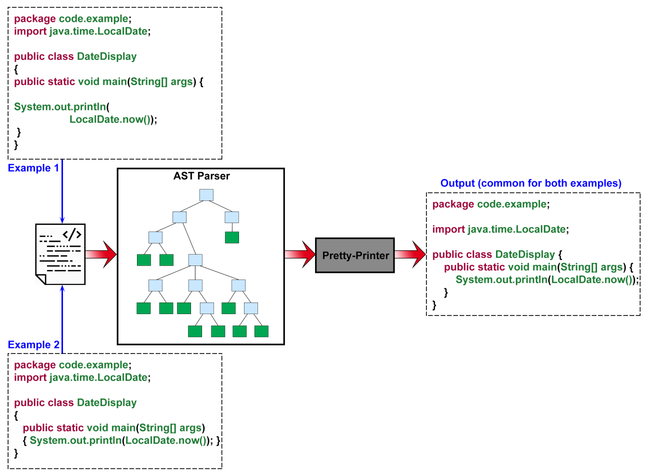


Figure 2: Simplified pipeline to quantify the student's ability to identify general concepts and break a problem into simpler tasks from the source code.



```
package code.example;
import java.time.LocalDate;

public class DateDisplay
{
public static void main(String[] args) {

System.out.println(
            LocalDate.now());
 }
}
```

```
package code.example;
import java.time.LocalDate;

public class DateDisplay
{
  public static void main(String[] args)
  { System.out.println(LocalDate.now()); }
}
```

```
package code.example;

import java.time.LocalDate;

public class DateDisplay {
    public static void main(String[] args) {
        System.out.println(LocalDate.now());
    }
}
```

Figure 3: Our software system parses the code, constructs an AST model, and then applies standard stylistic formatting conventions to generate output code that does not depend on the student's code-style.

how SLOC is measured. The code (formatting) style varies among programmers, and simply counting lines of code is not enough. Our parser constructs an abstract syntax tree (AST) from the program to address this issue. ASTs separate parsing (i.e., how the code is represented) from implementation (i.e., how the code is written). Nevertheless, we cannot count the number of lines directly from ASTs. First, we apply standard stylistic formatting conventions to every AST (i.e., pretty-printing [5]) and then capture the SLOC from the output source code (Figure 3).

3.3 Part-of-Speech Tagger/Analyzer

Figure 4 summarizes the next step, detecting the most common nouns. We begin by processing the final snapshots for all students. We then split the function names into component words (e.g., *play* and *game* for *playGame()*). The tokenization rules (i.e., camel case, snake case, etc.) are subject to the chosen programming language. Part-of-speech (POS) tagging is a popular NLP process that refers to marking up words in a corpus[1]. A POS tagger learns how to tag unlabeled data from a corpus of pre-annotated data and has high accuracy (95% or more). Most POS taggers are trained from treebanks in the newswire domain, such as the Wall Street Journal corpus of the Penn Treebank [18]. Tagging performance degrades on out-of-domain data [8]. To make matters worse, the lack of sentence structure in function names poses further challenges, and traditional approaches underperform [12].

Our solution takes advantage of domain-specific characteristics. On average, 66.5% of the function identifiers that we collected from various CS1 assignments had at least one verb and one noun, and the verbs were second-person singular 98% of the time. Unlike nouns, verbs are easily detectable. We considered only three tags (*noun*, *verb*, and *other*) and created an unsupervised model to locate the nouns from the position of the verbs. An entity can be singular or plural (i.e., functions *addBall* and *addBalls* refer to the same assignment-related entity *ball*). Therefore, our system singularizes the tokens [9]. Moreover, it ignores stopwords [6].

The model identifies the nouns in function names with 62.6% accuracy, much lower than the POS tagger performance on full-text articles. However, the goal is to identify the *main entities* which are the most frequently shared nouns among students. We process the

[1]There are at least eight tags for the main parts of speech: nouns, pronouns, adjectives, verbs, adverbs, prepositions, conjunctions, and interjections. POS Taggers often use more than thirty tags.

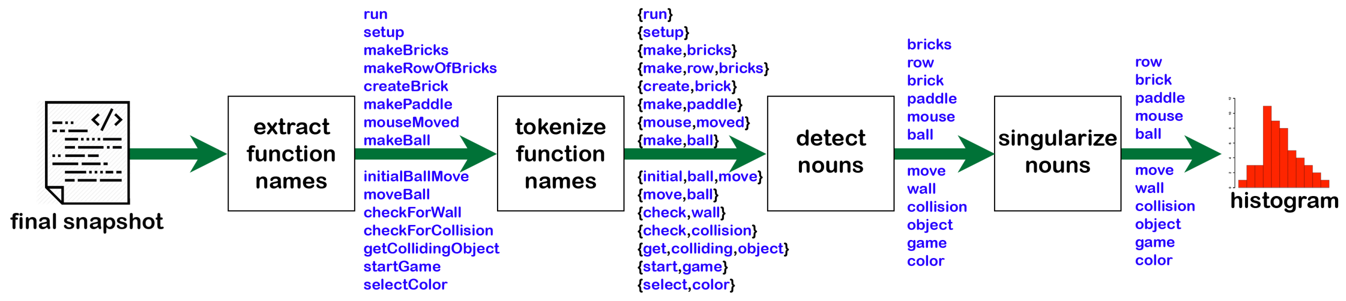


Figure 4: Example that illustrates how the main entities are detected. The part-of-speech tagger/analyzer first tokenizes the function names that are extracted from the *final program snapshot*. It then detects and singularizes the nouns. Finally, it aggregates the nouns from every student and creates a histogram with their total frequencies. The most popular nouns account for the main entities.

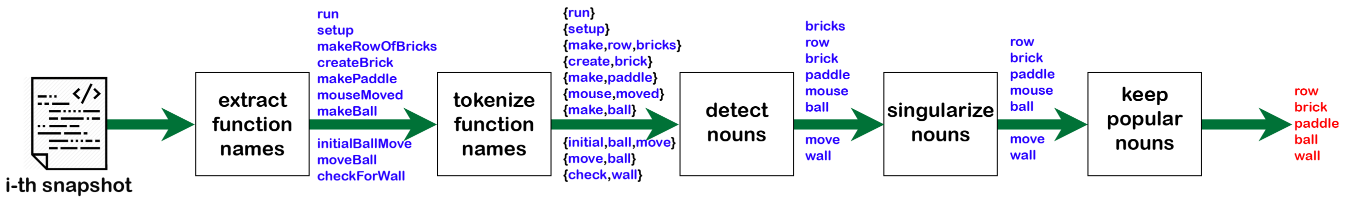


Figure 5: Example that illustrates how to count the main entities in a *random program snapshot*. The part-of-speech tagger/analyzer performs the same tasks as in Figure 4 to detect the nouns in the snapshot and then uses the histogram to keep only the popular nouns.

final snapshot for every submitted program and keep the nouns with popularity at least one standard deviation above the median. The model converges (i.e., identifies all main entities) as we process more student submissions (Figure 6). We also tried a supervised Bigram model (most function names consist of either two or three words). Although the accuracy increases to 73.4%, it does not make a difference to our end goal except for small datasets.

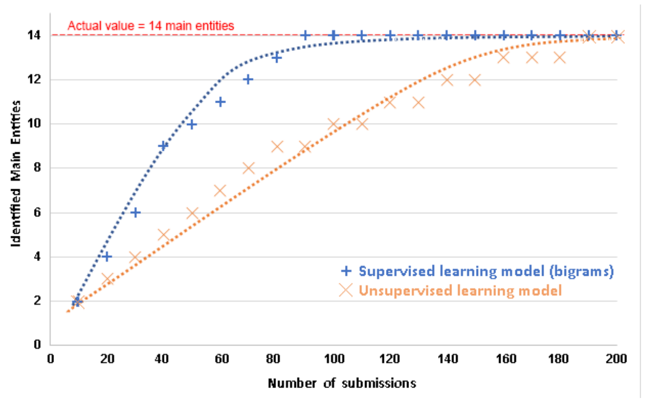


Figure 6: Both models converge to the full set of main entities. The supervised biagram model requires fewer submissions (N=100), but for N>190 the unsupervised model works just as well. The example is taken from the first assignment (*Breakout*) in our study.

3.4 Classifier

Once the main entities are identified, our system analyzes the program snapshots for every submission. We count the main entities found in the function names (Figure 5) and calculate the time spent on the assignment as an aggregate of relative timing information between a snapshot and the previous one, excluding breaks. A break occurs when two consecutive snapshots' timestamps differ by more than ten minutes. To quantify the student's ability to introduce early the main entities that are found in their final submission, we define the **entity progress ratio (EPR)** as the ratio of main entities in the last snapshot over the main entities in the first snapshot:

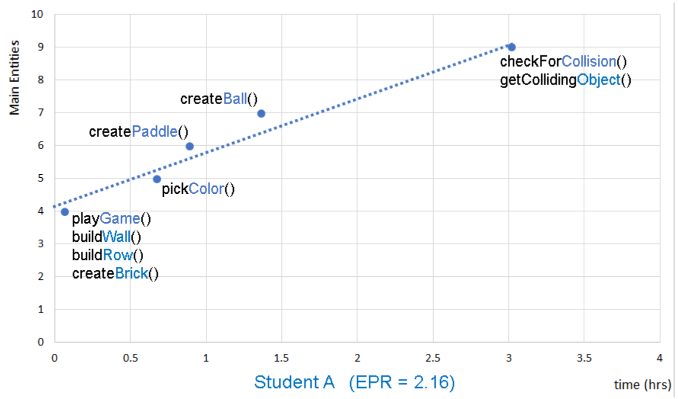
$$EPR = \frac{Main\ Entity\ Count_{\ last\ snapshot}}{Main\ Entity\ Count_{\ first\ snapshot}}$$

Both the numerator and denominator use linear regression fitted values. The mathematical expression for EPR is:

$$EPR = 1 + \frac{N \sum_{i=1}^{N} (E_i \cdot t_i) - \sum_{i=1}^{N} (E_i) \sum_{i=1}^{N} (t_i)}{\sum_{i=1}^{N} (E_i) \sum_{i=1}^{N} (t_i^2) - \sum_{i=1}^{N} (t_i) \sum_{i=1}^{N} (E_i \cdot t_i)}$$

where N is the number of snapshots, E_i is the number of main entities for the i-th snapshot, and t_i is the total time spent on the assignment.

Figure 7 uses an example to visualize the relationship between EPR and problem-solving approaches from different students. Lower EPR means that the student introduces earlier the main entities that appear in the final solution (better temporal decomposition).



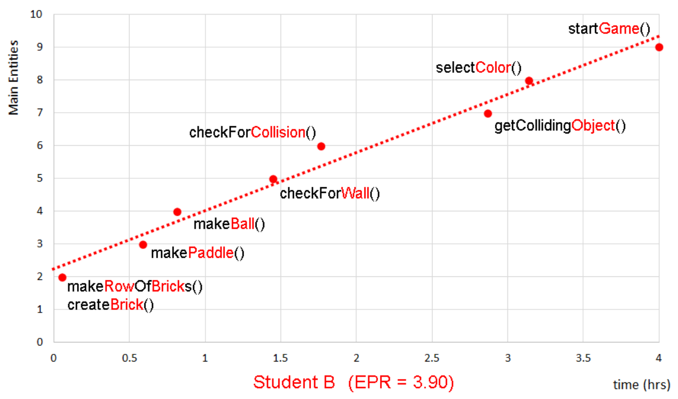


Figure 7: Example that visualizes how our method captures two student approaches for the same programming challenge [2]. We track when a main entity appears in a function name for the first time (*Wall* appears first in function *buildWall()* for student A and *checkForWall()* for student B). None of them follows a top-down or bottom-up approach but a mix of the two. However, student A creates a partial skeleton of the solution (i.e., function *playGame()* calls stub functions *buildWall(),buildRow()*, and *createRow()*, which are implemented later). Student B focuses on one subtask at a time. The respective EPR values suggest that student A has a broader perspective of the problem. *Note: Figure 4 and Figure 5 use student B as an example.*

4 EVALUATION

The EPR measurements are used in the last stage (*performance analysis*) to compare submissions (Figure 2). Each solution is evaluated on three criteria: 1) the time to develop it, 2) its complexity, and 3) the student's grades. To find the net time that the learner worked on the problem, we processed the snapshot timestamps. To capture the problem complexity, we used a) McCabe's cyclomatic complexity and b) the software lines of code (Section 3.2). For the student grades, we considered both the assignment grades and the score of the exam that is closer to the assignment (i.e., midterm for the first assignment and final exam for the second).

We clustered the students into two groups: i) those with EPR at least half a standard deviation below the median ($EPR \leq \widetilde{EPR} - \frac{1}{2}\sigma_{EPR}$) who exhibit signs of decomposition compared to their peers and ii) those with EPR at least half a standard deviation above the median ($EPR \geq \widetilde{EPR} + \frac{1}{2}\sigma_{EPR}$) who do the opposite. Table 1 summarizes our results.

The analysis suggests that program decomposition reduces substantially the time it takes to solve the problem. In both programming assignments, the students with low EPR finished much sooner than the other group. The difference in time between low EPR and high EPR is statistically significant (*p-value = 0.039* for the assignment before the midterm and *p-value = 0.027* for the assignment before the final exam). Our findings are in agreement with other researchers [13, 21] whereas the effect size is moderate. Though cyclomatic complexity and SLOC are not strictly measures of decomposition quality, high complexity correlates with opportunities for further decomposition. Piech et al. observed a higher correlation of the student development paths with their exam scores than their assignment grades [28]. They believe the underlying reason for this phenomenon is that the development path provides rich information regarding students' understanding of concepts beyond simply their final product. Our investigation backs up their explanation and also reveals a second reason. A weekly programming assignment imposes substantially more relaxed time constraints compared to a three-hour exam. As mentioned, decomposition helps solve a problem faster and therefore increases time efficiency leading to higher exam scores.

Table 1: How EPR affects the problem-solving time, the program complexity, and the student grades. The cyclomatic complexity (CYC) and the SLOC are per function. The assignment and the exam scores are normalized out of 100.

Student Group	$\overline{Time}$	$\overline{CYC}$	$\overline{SLOC}$	$\overline{Score}$	$\overline{Exam}$
	Assignment before midterm exam				
Low EPR	5.37 hrs	3.01	11.07	82	75
High EPR	6.12 hrs	3.35	12.27	80	62
All students	5.95 hrs	3.17	11.84	81	69
Standard Dev	2.64 hrs	0.68	2.55	9	23
P-score	0.039	0.004	0.034	0.049	0.028
Cohen's d	-0.284	-0.500	-0.471	0.222	0.565
	Assignment before final exam				
Low EPR	2.83 hrs	2.19	10.63	90	80
High EPR	3.75 hrs	2.49	11.03	88	73
All students	3.13 hrs	2.36	10.89	89	74
Standard Dev	1.72 hrs	0.47	2.69	10	14
P-score	0.027	0.018	0.067	0.077	0.049
Cohen's d	-0.535	-0.638	-0.149	0.200	0.500

5 THREATS TO RELIABILITY

We took a number of precautions to preserve the generality of our method:

- We emphasized objectivity. Our analysis relies on source code analysis rather than other, more subjective methods (e.g., surveys, think-aloud experiments, etc.).

- We used an unsupervised learning model to detect the main entities without prior knowledge of the programming assignment specifics.
- The source code parser does not depend on the programmer's formatting style.
- The metric to quantify decomposition (i.e., EPR) is normalized to allow comparison between students.

Despite our efforts, there are still threats to the reliability of our work. First, we observed frequently that the first snapshot contains only the main function. Therefore, even if the next includes a complete list of stub functions, the EPR value will be deceptively high. We found that in most cases, skipping the first 5% of the snapshots compensates for the early, transient development stage. Nevertheless, the threshold choice is arbitrary. Similarly for the threshold for the main entities (i.e., at least one standard deviation above the median popularity).

Second, we found that students with high EPR perform slightly better in the programming assignments than those with low EPR (Table 1). However, it remains unclear if the difference is statistically significant (i.e., p-score < 0.05). In the first program, the p-score is barely below 0.05 and in the second is 0.077. On the other hand, we can draw safer conclusions about the relationship between the EPR and the course exam score. A greater concern is the effect size. We used Cohen's d to calculate the magnitude of the difference between the two groups (i.e., low and high EPR). The effect size is moderate for the problem-solving time, the cyclomatic complexity, and the exam score and low for the lines of code and the assignment score.

Third, although the sample size (N=250 students) was sufficient to derive statistically significant results, a larger data set in terms of sample size and the number of assignments can be helpful for validation. It is reasonable to expect that the entities increase linearly with time. The two programming challenges we analyzed are quite different, yet the linear regression model fits both. Although the model selection is not the focal point in our work, considering more examples can increase our confidence in its generality.

6 DISCUSSION

What are the implications of our results for teaching and future research? The composition of the student body in CS1 is heavily diversified [27]. Predicting achievement includes many factors such as math background, programming experience, and previous academic performance [29]. Mastering program decomposition is no exception. Nevertheless, the distinction between correct and incorrect is not as apparent as in other cases (e.g., a program with a syntax error cannot even execute, an algorithmic mistake results in unexpected behavior, etc.).

Our findings establish that students who decompose their programs do not necessarily introduce fewer bugs and do not get substantially better assignment grades than their peers who do not. Therefore, many novice programmers with relatively strong background may still not fully comprehend the significance of decomposition. On the other hand, it becomes evident from our analysis that splitting complicated tasks into simpler subtasks acts as a catalyst for the program development. How can CS1 instructors convince the learners that decomposition is beneficial for everyone? One idea is to give out short yet time-constrained programming challenges in class. Under time pressure, many will jump directly into coding. On the other hand, one has to break down the challenge into smaller, simpler tasks to complete it on time.

Building automation tools around EPR can potentially lead to improved learning at scale. A software development tool (i.e., IDE) that calculates the EPR from the source code in real-time can notify the novice programmer to break down the problem into simpler tasks before the code becomes too complicated to handle. This is an example of a preventive intervention to keep the program complexity within reasonable levels. Inexperienced programmers tend to jump into coding without thinking about how to structure their programs around a simple design. Reminding the students to break down complex functions into subtasks is vital to ensure progress and avoid bugs. The logical next step to understand better how EPR can affect the programming time, the code complexity and the overall course performance is a research study with a treatment group (i.e., students who use the automation tool in an introductory programming class with a large audience).

7 CONCLUSION

Online courses and MOOC providers opened up learning opportunities to a wide audience and are now trying to keep up with the steep rise in enrollment. This paper presents a systematic approach to detect and quantify the student's ability to identify the main problem tasks and break them down into subtasks. It evaluates also its effects on the student's performance, the time it takes to solve a programming challenge and the complexity of the solution. Our findings suggest that the development path to the solution has an impact beyond the scope of the programming assignment itself.

Educators and researchers agree that program decomposition is a crucial software development skill that the novice learner must acquire. Nonetheless, it remains undetectable by the existing automated assessment tools (AATs), which analyze only the final product. We believe that the rapid advancements in machine learning and NLP opened up opportunities and hope that our work can contribute to more scalable solutions.

REFERENCES

[1] Khaled M. Alhawiti. 2014. Natural Language Processing and its Use in Education. *International Journal of Advanced Computer Science and Applications* 5, 12 (2014), 72–76. https://doi.org/10.14569/IJACSA.2014.051210
[2] Programming Assignment. CS1. "Breakout". https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/handouts/Assignment%203.pdf
[3] Programming Assignment. CS1. "Hangman". https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/handouts/Assignment%204.pdf
[4] Charis Charitsis, Chris Piech, and John 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* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 9 pages. https://doi.org/10.1145/3491140.3528269
[5] Wikipedia contributors. 2020. Prettyprint — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Prettyprint
[6] Wikipedia contributors. 2021. Stop word — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Stop_word
[7] Scott Crossley, Luc Paquette, Mihai Dascalu, Danielle S. McNamara, and Ryan S. Baker. 2016. Combining Click-Stream Data with NLP Tools to Better Understand MOOC Completion *(LAK '16)*. Association for Computing Machinery, New York, NY, USA, 6–14. https://doi.org/10.1145/2883851.2883931
[8] Jeffrey Ferraro, Hal Daume, Scott Duvall, Wendy Chapman, Henk Harkema, and Peter Haug. 2013. Improving performance of natural language processing part-of-speech tagging on clinical narratives through domain adaptation. *Journal*

of the American Medical Informatics Association 20, 5 (Mar 2013), 931–939. https://doi.org/10.1136/amiajnl-2012-001453

[9] Daniel Gildea and Daniel Jurafsky. 2002. Automatic Labeling of Semantic Roles. *Comput. Linguist.* 28, 3 (Sep 2002), 245–288. https://doi.org/10.1162/089120102760275983

[10] David Ginat. 2001. Starting Top-down, Refining Bottom-up, Sharpening by Zoom-In. *SIGCSE Bull.* 33, 4 (Dec 2001), 28–31. https://doi.org/10.1145/572139.572164

[11] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., USA.

[12] Suvarna G Kanakaraddi and Suvarna S Nandyal. 2018. Survey on Parts of Speech Tagger Techniques. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT).* 1–6. https://doi.org/10.1109/ICCTCT.2018.8550884

[13] Aaron Keen and Kurt Mammen. 2015. Program Decomposition and Complexity in CS1. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15).* Association for Computing Machinery, New York, NY, USA, 48–53. https://doi.org/10.1145/2676723.2677219

[14] Ulrich Kiesmueller, Sebastian Sossalla, Torsten Brinda, and Korbinian Riedhammer. 2010. Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods. 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, 274–278. https://doi.org/10.1145/1822090.1822167

[15] Tuomas Klemola and Juergen Rilling. 2003. A Cognitive Complexity Metric Based on Category Learning. In *Proceedings of the 2nd IEEE International Conference on Cognitive Informatics (ICCI '03).* IEEE Computer Society, USA, 106.

[16] Dharmender Singh Kushwaha and A. K. Misra. 2006. A Complexity Measure Based on Information Contained in the Software. In *Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems* (Madrid, Spain) *(SEPADS'06).* World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 187–195.

[17] James R. Leonard. 1991. Using a Software Engineering Approach to CS1: A Comparative Study of Student Performance. *SIGCSE Bull.* 23, 4 (Nov 1991), 23–26. https://doi.org/10.1145/122697.122700

[18] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (Jun 1993), 313–330.

[19] Sergey Matskevich and Colin S. Gordon. 2018. Generating Comments from Source Code with CCGs. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering* (Lake Buena Vista, FL, USA) *(NL4SE 2018).* Association for Computing Machinery, New York, NY, USA, 26–29. https://doi.org/10.1145/3283812.3283822

[20] Lawrence J. Mazlack. 1983. Introducing Subprograms as the First Control Structure in an Introductory Course. In *Proceedings of the Fourteenth SIGCSE Technical Symposium on Computer Science Education* (Orlando, Florida, USA) *(SIGCSE '83).* Association for Computing Machinery, New York, NY, USA, 265–270. https://doi.org/10.1145/800038.801062

[21] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[22] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Canterbury, UK) *(ITiCSE-WGR '01).* Association for Computing Machinery, New York, NY, USA, 125–180. https://doi.org/10.1145/572133.572137

[23] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of Programming in Scratch. 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, 168–172. https://doi.org/10.1145/1999747.1999796

[24] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE '07).* Association for Computing Machinery, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[25] David B. Palumbo. 1990. Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research* 60, 1 (1990), 65–89. https://doi.org/10.3102/00346543060001065 arXiv:https://doi.org/10.3102/00346543060001065

[26] Janice L. Pearce, Mario Nakazawa, and Scott Heggen. 2015. Improving Problem Decomposition Ability in CS1 through Explicit Guided Inquiry-Based Instruction. *J. Comput. Sci. Coll.* 31, 2 (Dec 2015), 135–144.

[27] Michaela Pedroni and Manuel Oriol. 2009. A comparison of CS student backgrounds at two universities. *CTIT technical reports series* 613 (2009).

[28] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12).* Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/2157136.2157182

[29] Nathan Rountree, Janet Rountree, Anthony Robins, and Robert Hannah. 2004. Interacting Factors That Predict Success and Failure in a CS1 Course. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE-WGR '04).* Association for Computing Machinery, New York, NY, USA, 101–104. https://doi.org/10.1145/1044550.1041669

[30] Daisuke Saito and Tsuneo Yamaura. 2014. Applying the top-down approach to beginners in programming language education. In *2014 International Conference on Interactive Collaborative Learning (ICL).* 311–318. https://doi.org/10.1109/ICL.2014.7017791

[31] Cynthia C. Selby. 2015. Relationships: Computational Thinking, Pedagogy of Programming, and Bloom's Taxonomy. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15).* Association for Computing Machinery, New York, NY, USA, 80–87. https://doi.org/10.1145/2818314.2818315

[32] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive Strategies and Looping Constructs: An Empirical Study. *Commun. ACM* 26, 11 (Nov 1983), 853–860. https://doi.org/10.1145/182.358436

[33] Raja Sooriamurthi. 2009. Introducing Abstraction and Decomposition to Novice Programmers. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09).* Association for Computing Machinery, New York, NY, USA, 196–200. https://doi.org/10.1145/1562877.1562939

[34] Yingxu Wang and Vincent Chiew. 2010. On the cognitive process of human problem solving. *Cognitive Systems Research* 11, 1 (2010), 81–92. https://doi.org/10.1016/j.cogsys.2008.08.003 Brain Informatics.

[35] Yingxu Wang and Jingqiu Shao. 2003. Measurement of the Cognitive Functional Complexity of Software. In *Proceedings of the 2nd IEEE International Conference on Cognitive Informatics (ICCI '03).* IEEE Computer Society, USA, 67.

[36] Daniela Zehetmeier, Axel Bottcher, Anne Bruggemann-Klein, and Veronika Thurner. 2019. Defining the Competence of Abstract Thinking and Evaluating CS-Students' Level of Abstraction. In *Proceedings of the 52nd Hawaii International Conference on System Sciences* (Honolulu, Hawaii, USA). 7642–7651. https://doi.org/10.24251/HICSS.2019.921