Detecting the Reasons for Program Decomposition in CS1 and Evaluating Their Impact

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 considered one of the four cornerstones of computational thinking, which is essential to software development [36]. It requires the ability to assess a problem at a high level, develop a strategy to combat it, and then design a solution. Our study focuses on the metacognitive aspect of decomposition. We try to understand the learner's thought process and, specifically, what makes the novice programmer decide to break down a function.

Researchers have studied decomposition in introductory programming courses through guided experiments, case studies, and surveys [23, 37]. In this work, we follow a different, more scalable approach. We develop an automated system to analyze 45,000 code snapshots from 168 students for a challenging CS1 programming assignment, detect the pivotal moments when they decide to decompose their programs, and identify what drives their decisions from the code. We then classify the students and study the relationship between the different categories, the code complexity, and the time to derive the final solution. We evaluate the impact of decomposition on the student's performance in the assignment and the course exams. Finally, we discuss the implications of our results for computing education.

CCS CONCEPTS

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

KEYWORDS

CS1, program decomposition, function decomposition, metacognition, complexity, student performance, computing education

ACM Reference Format:
Charis Charitsis, Chris Piech, and John C. Mitchell. 2023. Detecting the Reasons for Program Decomposition in CS1 and Evaluating Their Impact. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569763

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569763

1 INTRODUCTION

Decomposition is a higher-layer cognitive process that interacts with other cognitive processes such as abstraction, decision-making, inference, analysis, and synthesis. Therefore, it is perceived as one of the most challenging programming skills for learners to master [28]. In general, it can manifest on multiple levels in programming: data level, statement level, function level, module implementation, module interface, and system level [22].

In introductory programming courses where a program is primarily a single module, decomposition is synonymous with breaking down the overall task (i.e., main function) into subfunctions that implement simpler subtasks. In other words, decomposition manifests when the student introduces a function to simplify a task. Our study's primary goal is to identify what makes a student decide to decompose the code, which means two things. First, we must detect the pivotal moments when the novice programmer chooses to add a new function. Second, we need to compare the student program before and after to understand the driving force. This work tries to answer the following research questions:

(1) **What makes a student add a new function, and how can we detect the reason from the code?** (Section 3.4)
(2) **How can we classify student behavior from the identified reasons?** (Section 3.5)
(3) **How does the classification relate to a) the program complexity, b) the time it takes to derive the final solution, and c) the student's performance?** (Section 4)

We present a system to process program snapshots, identify the driving forces behind new functions, and then cluster these reasons to profile the student's behavior. Next, we explore how they are related to a) the program complexity, b) the time to complete the programming challenge, and c) student performance. We proceed with threats to the reliability of our work. Finally, we share our thoughts on the implications for computing education and conclude with a summary and final remarks.

2 RELATED WORK

Researchers consider problem decomposition a critical skill in software development. Many examine the differences in its uses and strategies between experts and novices [14, 30]. Ginat studies the various perspectives of problem decomposition [9]. Wing identifies breaking problems down by functionality as part of computational thinking [33].

Keen and Mammen use the cyclomatic complexity [21] to compare the decomposition skills students develop during a term-long project against those acquired by their cohorts in courses with stand-alone projects [16]. Charitsis et al. use NLP for a system that learns the main problem-related entities and tracks how they evolve

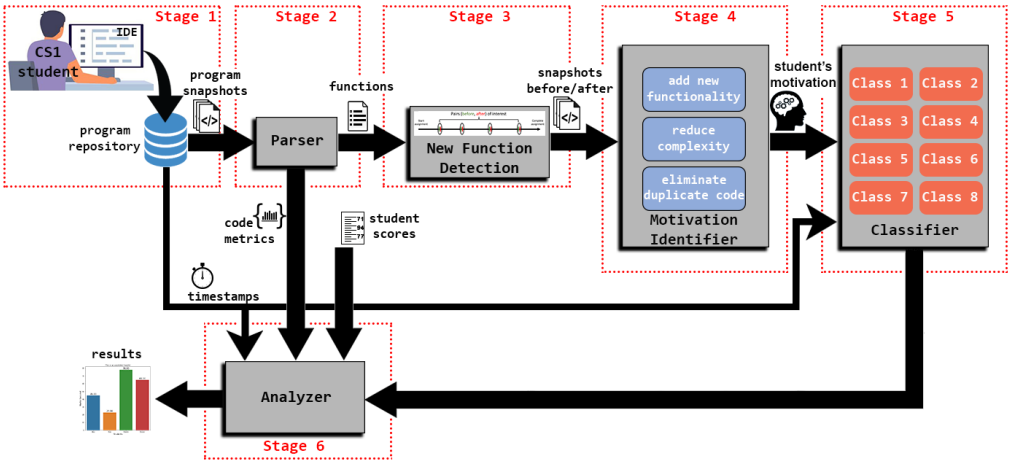

Figure 1: Simplified pipeline to classify the students based on when and why they decide to decompose their code.

to quantify the learner's ability to break down the problem into simpler tasks [3]. Haglund et al. emphasize controlling complexity and investigate how well students in CS1 create and maintain abstractions through user-defined functions [11].

Students with high metacognitive awareness perform better than those with low self-efficacy and underdeveloped metacognitive abilities [5, 7, 20]. Yadav et al. study the relationship between computational thinking and metacognition in K-12 classrooms [34]. Parham et al. explore how computer science students solve problems to understand the relationship between metacognition and schemata [25].

Researchers proposed several methods to assess students' decomposition abilities [6, 10, 19]. Although surveys, think-aloud experiments, and guided-based research techniques are undeniably insightful, applying them at scale takes a lot of effort. 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 program structure. Doing so by hand is inefficient and time-consuming. This paper presents a computer-assisted approach to studying the relationship between decomposition and program complexity, the time to derive the solution, and student performance.

This work makes several contributions to early-stage research [2]: 1) it explores how the factors that drive decomposition relate to code complexity and student performance, 2) it delves into subtle concepts and addresses threats to the method's reliability, and 3) it discusses the implications for computing education.

3 METHOD

Figure 1 summarizes the steps of our method, which begins with the data collection. We gather multiple student snapshots during the program development (stage 1). Next, we parse them to find where a function is introduced (stage 2), locate the pre- and post-decomposition snapshots (stage 3), and compare those two to understand what drives the student's decision (stage 4). We repeat this process for each user-defined function to profile the decomposition behavior (stage 5). The following subsections elaborate on each step.

3.1 Data Collection

We modified the Integrated Development Environment (IDE) that CS1 learners use to develop their programs to commit a source code snapshot to a local repository every time they save or run the program. When students submit online the assignment solution, they can opt-in to also upload the repository with the snapshots taken in their problem-solving journey. We collected 49,000 program snapshots extracted from 168 student submissions for this research study. We filtered out programs with syntax errors that did not compile and analyzed the remaining snapshots (i.e., approximately 45,000). The programming challenge is structured such that the deliverable tasks are well-defined [31]. The novice programmers 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 individual who is often unsure how to begin. Nevertheless, the program decomposition is left to the student. The first stage completes upon extracting the compilable snapshots and their timestamps from the local repository.

3.2 Parser

The parser processes the snapshots in the local repository and performs three tasks: 1) it creates a list with the function signatures, 2) it tracks the calls to those functions, and 3) it captures source code-related metrics to evaluate decomposition (Section 4). There are many code-based software complexity measures [12, 17, 18, 21, 32]. Like Keen and Mammen [16], we used McCabe's cyclomatic complexity metric and tracked the software lines of code (SLOC).

The formatting style varies among programmers, and simply counting lines of code is not enough. The parser constructs an abstract syntax tree (AST) from the program to ensure that the style does not affect the results. 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 [4]) and count the lines from the output source code.

3.3 Decomposition Detection

In CS1, programs consist primarily of a single module. Therefore, decomposition is synonymous with breaking down functions into subfunctions to simplify a task. Figure 2 summarizes the next step in our method. One must look for places in the code where a function is introduced to detect decomposition. Moreover, we need to ensure that it gets eventually called from somewhere else in the program (i.e., it is part of the execution flow).

First, we collect the student submissions, extract the program snapshots (Section 3.1), and process the source code to retrieve the functions for every snapshot (Section 3.2). For each function, we find the last snapshot before it gets declared (Figure 2b)) and the one that first calls it[1] (Figure 2c)). We need them to determine what triggered the decomposition (Section 3.4).

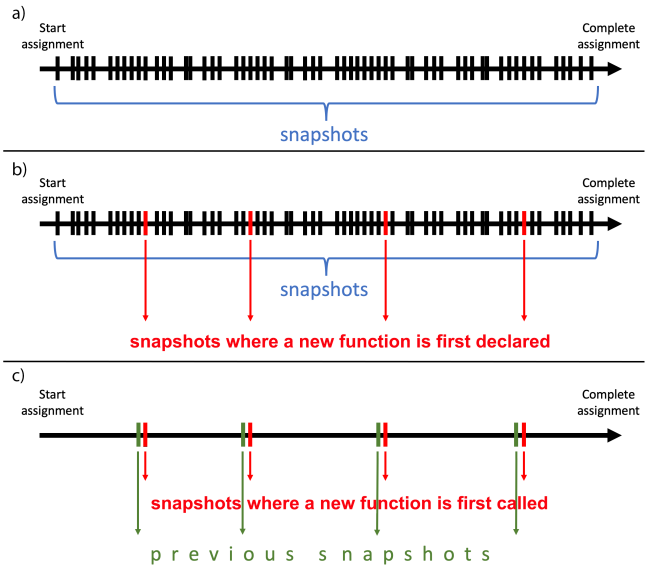

Figure 2: Steps to detect decomposition. We need to: a) collect snapshots during the program development, b) find the snapshot where a function is first declared, c) locate the previous snapshot and then the first that calls it (i.e., after declaration).

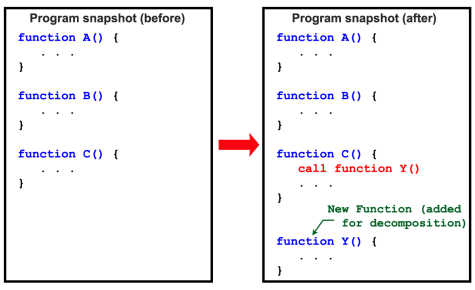

Figure 3: We consider only cases where two subsequent snapshots differ by only one function. All but one function signature must be identical in both snapshots, although the internal implementations may vary.

Although it is not very common, a function may be introduced while another is removed. In this case, both changes affect the program, and one cannot draw safe conclusions regarding the impact of the new function alone. Therefore, our analysis considers only cases where two subsequent snapshots differ by only one function (Figure 3).

3.4 Motivation Behind a New Function

Next, we address the first research question: *What makes a student add a new function, and how can we detect the reason from the code?* There are three reasons to introduce a function.

First, the novice programmer may want to either a) insert new code to break a bigger problem into smaller chunks and, as a result, inject stub functions to lay out the steps or b) add new functionality (Figure 4).

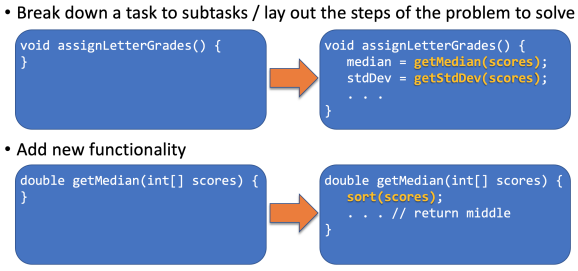

Figure 4: In the example on top, the student lays out the steps to assign letter grades by calling two stub functions to get the median and the standard deviation. On the bottom, the student adds the implementation for the first stub function that calculates the median and, in doing so, introduces a helper function to sort the scores.

Second, as students work on implementing a given task, they keep adding code. At some point, they may realize that a function has grown long or is too complicated to handle and decide to split it to make progress. Modifying the program in a way that does not alter its external behavior but improves the internal structure of the code is a process known as *refactoring* [8, 24] (Figure 5).

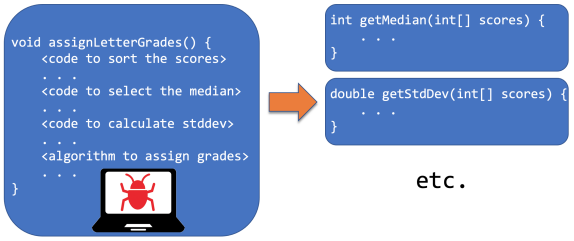

Figure 5: In this example, the student decides to split a long and complex function into simpler subfunctions.

[1] In most cases, the snapshot that declares a function is also the one that first calls it. If not, we must search in the subsequent snapshots to find the one that first calls it.

Third, the student may want to transfer a code fragment that appears similar in multiple locations[2] to a *shared* function to eliminate duplication (Figure 6).

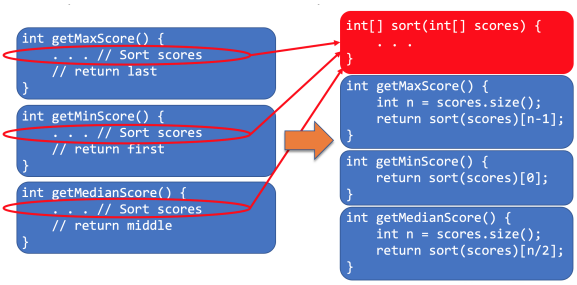

Figure 6: In this example, the student decides to introduce a function that sorts the scores to avoid duplication in the code that calculates the maximum, minimum, and median scores.

How can we identify the motivation from the code?

Novices often save the code with syntax errors to protect their work. The system ignores non-compilable snapshots, finds where a function is first declared (Figure 2b), and, from there, locates the previous snapshot (Figure 2c). Then it compares this *before-decomposition* with the first five *after-decomposition snapshots*[3] (Figure 7) to determine what motivated the student.

3.5 Classifier

A programmer swaps hats frequently between various activities. For example, one can start by trying to add a new function and then realize it becomes easier if the code is structured differently. The system detects the motivation behind every new function (Figure 1/stage 4), and then the classifier uses this information to profile the student behavior (Figure 1/stage 5). There are eight taxonomy groups based on the programmer's motivation (Figure 8).

Functions that *add functionality* grow over time. Thus, one can track the size to detect them (Table 1). Noting the time when this happens is equally important. A student is considered to add new functionality *early* if most size-growing functions appear before the development midpoint (i.e., 50% progress). The system uses the snapshot timestamps (Section 3.1) to find the midpoint. It calculates the net 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 timestamps differ by ten minutes [35].

The intention to remove duplicate code (known as *clones*) is easily detected as the student moves the common code to a new function that calls multiple times (Table 1). Clone elimination qualifies as motivation for decomposition if 10% or more of the functions are shared (i.e., called more than once). Finally, the desire to reduce the program complexity qualifies as a reason if at least 10% of the functions come from splitting longer ones (i.e., refactoring). Both thresholds are relatively relaxed and, therefore, safe to use.

[2]Roy et al. provide a clone classification involving the amount and the way a code fragment is duplicated [27].

[3]We consider five snapshots instead of one because implementing a function is incremental (i.e., takes some time), not instant. In addition, there are two requirements: 1) no other function is removed or added, and 2) the newly declared function is called.

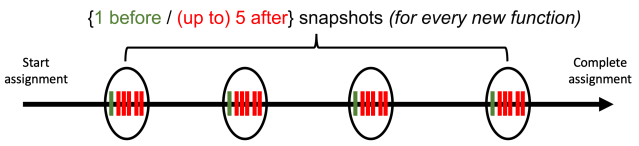

Figure 7: Function implementation is not an instant but an incremental process. Therefore, we consider up to five snapshots after a function is introduced.

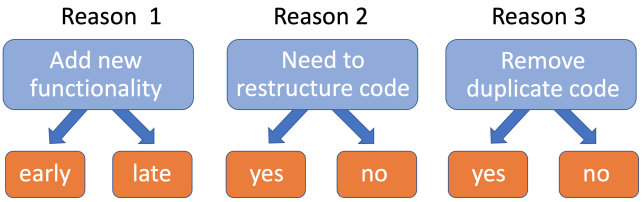

Figure 8: There are three reasons to add a new function. Our classifier considers two options for every reason. Thus, there are 2^3=8 taxonomy groups.

Table 1: The manifestation of the student's motivation in the program before and after the decomposition.

Student's motivation	Manifestation in the code
Add new functionality	A new function is added and the initial function (now the caller function) grows in size.
Need to restructure code	Code moves from a long function to a new function that the initial (long) function now calls. Thus, the caller/initial function shrinks.
Remove duplicate code	The new function is called multiple times (i.e., the common code moved to a shared function which is called in multiple places in the program)

4 RESULTS

We collected 49,000 program snapshots extracted from 168 students for a challenging CS1 programming assignment. After filtering out non-compilable code, we analyzed approximately 45,000 snapshots. In our study, we tried to answer a series of compelling questions.

- Which is the primary reason that makes students decompose their programs?

Adding functionality is common for all students. The analysis suggests that 54.2% add most functions late. Moreover, it reveals that 43.4% of learners feel the need to restructure their programs, and 37.5% identify and remove duplicate code (Figure 9).

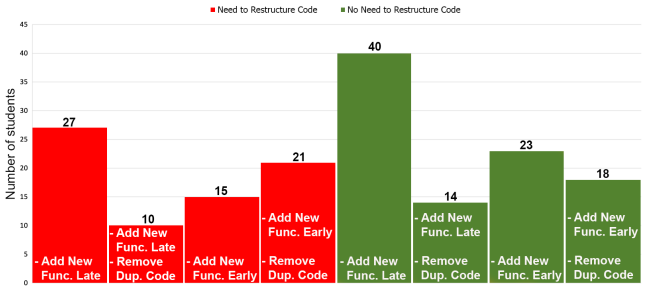

Figure 9: Classification histogram for the students who completed the programming challenge

- How does each reason relate to program complexity?

We measured McCabe's cyclomatic complexity and tracked each program's software lines of code (SLOC) (Section 3.2). We were interested only in the snapshots before and after the decomposition and grouped them based on what triggered it. Finally, we calculated the median values for the cyclomatic complexity (Figure 10) and the SLOC (Figure 11) in each group.

Students who need to restructure their code have the most significant drop in cyclomatic complexity and lines of code, 7.4% and 7.3%, respectively. On the other hand, adding gradually new functionality means that complexity is under control. Thus, it decreases only by 3.8%. SLOC falls even less, by 1.3%. Table 2 shows the mean, median and standard deviation for the complexity and the SLOC before and after the decomposition. The results are statistically significant (i.e., the p-value is 0.004 for the cyclomatic complexity and 0.005 for the SLOC).

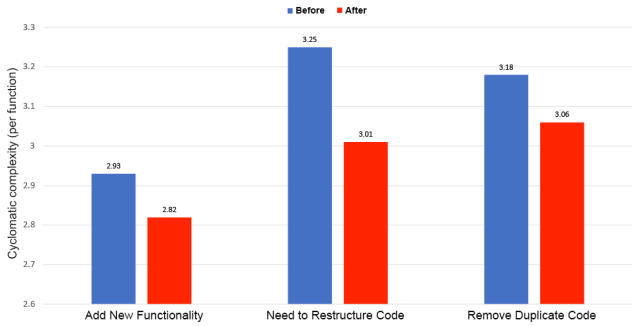

Figure 10: The cyclomatic complexity (per function) before and after decomposition depending on the motivation.

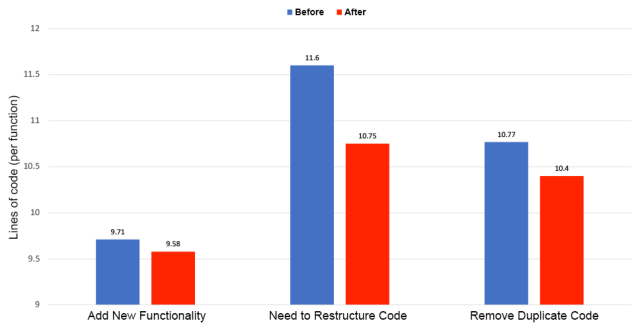

Figure 11: The SLOC (per function) before and after decomposition depending on the motivation.

Table 2: Statistical measures for the reasons that make a student add a function, before and after the decomposition.

	Add new functionality	Need for code restructure	Remove duplicate code
Before the new function			
CYC Median	2.93	3.25	3.18
CYC Mean	2.94	3.72	3.22
CYC Std Dev	0.37	0.41	0.39
SLOC Median	9.71	11.60	10.77
SLOC Mean	9.81	12.94	11.13
SLOC Std Dev	1.20	1.43	1.15
After the new function			
CYC Median	2.82	3.01	3.06
CYC Mean	2.86	3.14	3.13
CYC Std Dev	0.37	0.39	0.38
SLOC Median	9.58	10.75	10.40
SLOC Mean	9.83	11.08	10.87
SLOC Std Dev	0.89	1.07	0.91

- What is the impact on the time to derive the solution?

We tracked the total time to complete the programming challenge for each student group. Then we performed a pairwise comparison between groups that restructured/did not restructure the program (Figure 12). The main takeaway is that students who need to refactor their code take, on average, 17% more time to deliver the final solution than their peers who don't. Adding new functionality early in the development helps reduce the overall time, although only by 6%. The p-value for the average time is 0.005.

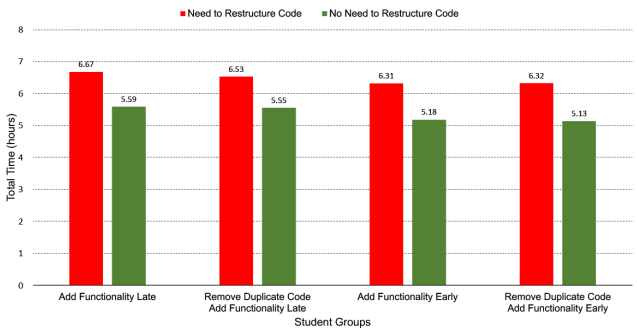

Figure 12: Total time (mean value) to deliver the final solution for each group. The groups are paired so that the reasons that drive decomposition differ only in terms of the need to restructure the code. In red are the students who feel the need to restructure the code and in green are those who don't.

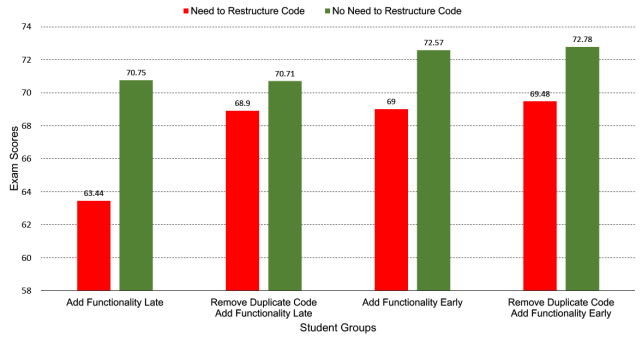

Figure 13: The exam score (mean value) for each student group. The score range is 0 to 100, the mean is 70, the median is 71 and the standard deviation is 3.5. The groups are paired, so that the only difference is the need to restructure the code.

-How does each group perform?

We considered the assignment grades and the exam scores to evaluate student performance. For each group, we compared the mean values. The assignment grades are almost unaffected. On the other hand, exam scores fluctuate. Our findings agree with other studies that observed a higher correlation between student programming skills and exam scores than assignment grades [3, 26]. A pairwise comparison between groups that restructured/did not restructure the program indicates that the latter performed better in the exams (Figure 13). The p-value is 0.006, slightly higher than the statistical significance threshold of 0.005.

5 THREATS TO RELIABILITY

We took several precautions to preserve the research's generality and objectiveness. Our work relies on source code analysis rather than other, more subjective methods (e.g., surveys, think-aloud experiments, etc.). Moreover, it considers multiple metrics (i.e., cyclomatic complexity and SLOC) to capture the program complexity and ensures that the measurements do not depend on the programmer's formatting style. Finally, to determine how early a student introduces new functionality, we use timestamps and consider only the net time the student has worked on the assignment by that point.

Despite our efforts, there are still threats to the reliability of our work:

(1) Detecting decomposition does not include rare cases where two subsequent snapshots differ by more than one function (Section 3.3).
(2) We attribute the function decomposition to a single reason (Section 3.4). Although rare, there can be multiple driving forces. For example, the student may want to reduce the program complexity and, at the same time, remove duplicate code.
(3) The system considers up to five snapshots after the decomposition to determine its impact on the program (Section 3.4). This number is sufficient in most cases to account for the time it takes to complete the new function's implementation. Nevertheless, the selection is still arbitrary.
(4) Similarly, for the threshold (i.e., *development midpoint*) to determine if a student adds functionality early and the percentage (i.e., *10% or more*) of shared functions that make duplicate code elimination qualify as a reason for decomposition (Section 3.5).
(5) Although the sample size (i.e., *N=168 students*) is not small, a larger data set of students and assignments can be helpful for validation.

6 DISCUSSION

The analysis suggests that students who need to restructure the code fail to keep the program complexity under control and spend 17% more time deriving the final solution. Although it is not immediately evident that simple metrics for measuring code complexity are related to the cognitive obscureness encountered by the programmer, code complexity is a proxy for increasing difficulty. The assignment grades and exam scores can be used to study the short-term and long-term effects, respectively [15]. A weekly programming assignment imposes more relaxed time constraints than a three-hour exam. As mentioned, lower complexity is correlated with solving a problem faster. Thus, it increases time efficiency leading to higher exam scores.

Arguably the most substantial implications of our work for CS education lie in the field of *metacognition*, as novice students often lack a deep understanding of how they learn. A growing research community advocates assisting the novice in building a mental scaffold around which they can correctly place knowledge and, in doing so, develop metacognitive awareness [1, 5, 7, 13, 20, 25, 29, 34]. Top universities and educational institutions use one-on-one sessions between the learner and the TA to provide formative feedback on how students implement programming methodology and have developed software to facilitate the discussion [35]. Exploring when students decompose the program, what drives their decision, how complicated their code is at the time, and to what extent they have an overall plan to add functionality gradually can help them develop a growth mindset. It seems reasonable to believe that some ideas in this paper can be incorporated into computer education software to foster metacognition.

7 CONCLUSION

Educators and researchers agree that program decomposition is a fundamental software development skill that the novice learner must acquire. This paper presents a systematic approach to detecting when novice programmers decompose their code and classify their behavior from the motivation that drives their decision. We built a system to perform these steps and used it to analyze the results in a CS1 programming challenge. Our findings suggest that students often introduce a function to add functionality, while many try to restructure their code to manage program complexity. Those who keep complexity under control without code refactoring derive the final solution faster. Investigating when and why students decide to decompose a function can help them acquire a deeper understanding of the problem-solving journey around which they can correctly place knowledge and, in the process, develop metacognitive awareness.

REFERENCES

[1] Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the Role of Self-Regulated Learning on Introductory Programming Performance. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) *(ICER '05)*. Association for Computing Machinery, New York, NY, USA, 81–86. https://doi.org/10.1145/1089786.1089794

[2] Charis Charitsis, Chris Piech, and John Mitchell. 2022. Profiling Program Decomposition in CS1. In *EDULEARN22 Proceedings* (Palma, Spain) *(14th International Conference on Education and New Learning Technologies)*. IATED, 10128–10135. https://doi.org/10.21125/edulearn.2022.2447

[3] 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

[4] Wikipedia contributors. 2022. Prettyprint — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Prettyprint

[5] S. Coutinho. 2008. Self-efficacy, Metacognition, and performance. *North American Journal of Psychology* 10 (03 2008), 165–172.

[6] Fadi P. Deek, Starr Roxanne Hiltz, Howard Kimmel, and Naomi Rotter. 1999. Cognitive Assessment of Students' Problem Solving and Program Development Skills. *Journal of Engineering Education* 88, 3 (1999), 317–326. https://doi.org/10.1002/j.2168-9830.1999.tb00453.x

[7] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3364510.3366170

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[9] David Ginat. 2002. On Varying Perspectives of Problem Decomposition. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 331–335. https://doi.org/10.1145/563340.563470

[10] Brian Thomas Gong. 1988. *Problem Decomposition by Computer Programmers*. Ph. D. Dissertation. Stanford, CA, USA.

[11] Pontus Haglund, Filip Stromback, and Linda Mannila. 2021. Understanding Students' Failure to use Functions as a Tool for Abstraction - An Analysis of Questionnaire Responses and Lab Assignments in a CS1 Python Course. *Informatics in Education* 20, 4 (2021), 583–614. https://doi.org/10.15388/infedu.2021.26

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

[13] Matthias Hauswirth and Andrea Adamoli. 2017. Metacognitive Calibration When Learning to Program. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '17)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3141880.3141904

[14] Chun-Heng Ho. 2001. Some phenomena of problem decomposition strategy for design thinking: differences between novices and experts. *Design Studies* 22 (2001), 27–45.

[15] Petri Ihantola and Andrew Petersen. 2019. Code Complexity in Introductory Programming Courses. https://doi.org/10.24251/HICSS.2019.924

[16] 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

[17] 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.

[18] 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.

[19] Kyungbin Kwon and Jongpil Cheon. 2019. Exploring problem decomposition and program development through block-based programs. *International Journal of Computer Science Education in Schools* 3, 1 (Apr. 2019), 3–16. https://doi.org/10.21585/ijcses.v3i1.54

[20] Murali Mani and Quamrul Mazumder. 2013. Incorporating Metacognition into Learning. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/2445196.2445218

[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] Steve McConnell. 1993. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, USA.

[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] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph. D. Dissertation. https://www.proquest.com/dissertations-theses/refactoring-object-oriented-frameworks/docview/303983132/se-2 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2022-02-27.

[25] Jennifer Parham, Leo Gugerty, and D. E. Stevenson. 2010. Empirical Evidence for the Existence and Uses of Metacognition in Computer Science Problem Solving. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 416–420. https://doi.org/10.1145/1734263.1734406

[26] 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

[27] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[28] 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

[29] Teresa M. Shaft. 1995. Helping Programmers Understand Computer Programs: The Use of Metacognition. *SIGMIS Database* 26, 4 (Nov 1995), 25–46. https://doi.org/10.1145/223278.223280

[30] Ting Song and Kurt Becker. 2014. Expert vs. Novice: Problem Decomposition/Recomposition in Engineering Design. In *2014 International Conference on Interactive Collaborative Learning (ICL)*. IEEE, Dubai, United Arab Emirates, 181–190. https://doi.org/10.1109/ICL.2014.7017768

[31] Stanford. CS1. Programming assignment: Breakout. https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/handouts/Assignment%203.pdf

[32] 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.

[33] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (Mar 2006), 33–35. https://doi.org/10.1145/1118178.1118215

[34] Aman Yadav, Ceren Ocak, and Amber Oliver. 2022. Computational Thinking and Metacognition. *TechTrends* 66, 3 (2022), 405–411. https://doi.org/10.1007/s11528-022-00695-z

[35] Lisa Yan, Annie Hu, and Chris Piech. 2019. Pensieve: Feedback on Coding Process for Novices. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 253–259. https://doi.org/10.1145/3287324.3287483

[36] Irma Yuliana, Langga Putra Octavia, and Endah Sudarmilah. 2020. Computational thinking digital media to improve digital literacy. *IOP Conference Series: Materials Science and Engineering* 821, 1 (Apr 2020), 012032. https://doi.org/10.1088/1757-899x/821/1/012032

[37] 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