Understanding and Measuring Incremental Development in CS1

Anshul Shah
ayshah@ucsd.edu
University of California, San Diego

Michael Granado
magranado@ucsd.edu
University of California, San Diego

Mrinal Sharma
m1sharma@ucsd.edu
University of California, San Diego

John Driscoll
jjdrisco@ucsd.edu
University of California, San Diego

Leo Porter
leporter@eng.ucsd.edu
University of California, San Diego

William G. Griswold
wgg@eng.ucsd.edu
University of California, San Diego

Adalbert Gerald Soosai Raj
gerald@eng.ucsd.edu
University of California, San Diego

ABSTRACT

Incremental development is the process of writing a small snippet of code and testing it before moving on. For students in introductory programming courses, the value of incremental development is especially higher as they may suffer from more syntax errors, lack the proficiency to address complicated bugs, and may be more prone to frustration when struggling to correct code. However, to evaluate the effectiveness of interventions that aim to teach programming processes such as incremental development, we need to develop measures to assess such processes. In this paper, we present a way to measure incremental development. By qualitatively analyzing 15 student coding interviews, we identified common behaviors in the programming process that relate to incremental development. We then leveraged a dataset of over 1000 development sessions – about 52,000 code snapshots at compilation time – to automatically detect the common behaviors identified in our qualitative analysis. Finally, we crafted a formal metric, called the "Measure of Incremental Development" (MID), to quantify how effectively a student used incremental development during a programming session. The MID detects common non-incremental development patterns such as excessive debugging after large additions of code to automatically assess a sequence of snapshots. The MID aligns with human evaluations of incrementality with over 80% accuracy. Our metric enables new research directions and interventions focused on improving students' development practices.

CCS CONCEPTS

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

KEYWORDS

incremental development, automated assessment, CS1, programming process

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569880

ACM Reference Format:
Anshul Shah, Michael Granado, Mrinal Sharma, John Driscoll, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2023. Understanding and Measuring Incremental Development in CS1. 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/10.1145/3545945.3569880

1 INTRODUCTION

Based on conversations with practitioners at the SIGCSE Technical Symposium, it appears that many CS1 instructors have had a student who comes to their office hours after writing all the code for an assignment, but the student would not have compiled or run their program even once. Instructors would likely recognize that the student's strategy to write the whole assignment in one attempt is an example of an ineffective programming process.

The computing education field places high importance on students learning effective programming processes. One such process is incremental development—the process of writing a small snippet of code and testing the snippet before proceeding. In fact, a "Fundamental Programming Construct" in the Computing Curricula 2001 report is that "students apply the techniques of structured decomposition to break a program into smaller pieces" [23].

Although the technique is a key learning goal for computing curricula, incremental development is not precisely defined and able to be assessed at scale by educators. "Incremental development" is generally understood as a "code-a-little, test-a-little" strategy [1, 2], but such general definitions present ambiguity. For example, what is considered "a little" amount of code? How much code should a student add in one step? Even with a precise definition, educators simply cannot observe every student's development process to detect incremental development.

However, an automated metric that monitors a student's adherence to incremental development with comparable accuracy to an instructor could enable every student to receive feedback on their development process. To thoroughly understand incremental development with the goal of automatically measuring it, we ask the following research questions:

RQ1: What common programming behaviors characterize incremental development among CS1 students?

RQ2: How can we measure incremental development by creating a metric to automatically detect these common programming behaviors?

To answer RQ1, we conducted programming interviews with CS1 students and qualitatively analyzed these interviews to identify common patterns and behaviors in a student's programming process. For RQ2, we collected thousands of student code snapshots from three programming assignments in a CS1 course to automate the detection of the common behaviors we observed in our qualitative analysis. We synthesize our findings to develop a metric to automatically measure incremental development among introductory computer science students at scale.

2 RELATED WORK

Several early works in computing education research have provided qualitative, process-oriented assessments of student programming behavior. Soloway in 1988 published "Studying a Novice Programmer" that presents a variety of qualitative methods that aim to understand novices' reasoning, approaches, and debugging skills [22]. Perkins et al. in 1986 use clinical studies of children's programming behaviors to verify that novices often have trouble decomposing a program into smaller pieces and may haphazardly tinker with their code while debugging [18]. In fact, a recent literature review on process-oriented approaches to analyze novice programming pointed to numerous studies that use the approach of observing code snapshots during student development to detect patterns in compilations, problem-solving, debugging, etc. [24].

Recent work has focused on automated, quantitative assessments of the development process. A trial of work followed Jadud's initial work to define the Error Quotient (EQ) in 2006, which measures how effectively a student recovers from a syntax error [11]. The Watwin Score [25], Repeated Error Density [3], and the Normalized Programming State Model [6] all built on the idea of examining features of the development process to predict student outcomes and explain the variation in student achievement. However, as Kazerouni [13] points out, none of these metrics actually measure incremental development. A recent metric by Charitsis [7] uses NLP techniques to measure program decomposition among novice Java programmers. However, as we explain in Section 7, this approach measures a different aspect of problem-solving from the metric we developed.

Based on our goal, the works by Kazerouni et al. [12–14] stand out as the most similar to our approach. Kazerouni et al. present four metrics to measure the level of incremental development and procrastination in a programming project in a large-scale, advanced Data Structures and Algorithms course [14]. These metrics rely on the idea of "wall-clock" time and reward students for working on a project in consistent time increments instead of concentrating the work at the end of the project timeline. For example, in the Incremental Checking metric [14], students are rewarded for consistently making small edits with little time between compilations.

Kazerouni's metrics capture a dimension of incremental development related to incremental *project management*—the idea of spreading out the work on a project as it relates to the due date. The metrics evaluate a development session as non-incremental when a student writes significant solution code and many unit tests close to a deadline. However, our work in this project defines incremental development only by the developmental pattern of the code itself, regardless of the due date or overall project progress. With our framing of incremental development, a student can use a non-incremental process days before the deadline or an incremental process the night of the deadline. We aim to present a metric that can be leveraged in parallel with Kazerouni's metrics to measure fine-grained incremental development on an individual module level and a more general project management level.

3 METHODS

3.1 RQ1: Conducting Student Coding Interviews

To observe the behaviors associated with incremental development, we conducted 15 interviews over Zoom in accordance with our approved Human Subjects protocol. The participating students were selected halfway through a 10-week CS1 course taught at our research-intensive university in the Spring 2021 quarter. In the interview, we asked students to complete the Rainfall Problem [21], which asks students to write one function in Python that finds the average of all the positive values in a list. We allowed students to access online documentation while completing the problem and to ask clarifying questions to the interviewer. We also asked students to either think-aloud while programming or answer questions about their process at the end of the interview using a process called stimulated recall [4].

Analysis of Student Coding Interviews: We analyzed these interviews by taking note of the changes they made to their code and the timestamp of those changes to identify common patterns in the students' development. We took a snapshot of a student's code at each run to get a sense of how frequently they were testing their code. Using these snapshots, we extracted features such as how many lines were added and deleted between compilations, how many total snapshots were needed to reach a solution, and how frequently a student ran into syntactic or semantic errors. We then created clusters of students based on the similarity of these features and common patterns among the coding sessions, which we will describe in Section 4.1.

3.2 RQ2: Analyzing Student Code Snapshots

To craft a metric that automates the manual process of detecting incremental development, we collected student code snapshots from three programming assignments in a CS1 course taught in Spring 2022 at our university. A total of 481 students were enrolled in the course, which included 8 weekly assignments written in Python. The three assignments [20] we analyzed were developed by the research team and included multiple interacting functions to elicit a range of incremental development behaviors by students.

- Assignment 1 – "Fun with Tweets" assigned in Week 6 – includes one function to find the number of words in a tweet, a second function to find the average tweet length in a list of tweets, and a third function that returns all tweets in a list that are shorter than the average tweet length.
- Assignment 2 – "COVID Clusters" assigned in Week 8 – includes one function to create a dictionary with per-capita COVID-19 cases for each state from a list of tuples, another function to categorize states as low, medium, or high COVID-19 risk, and a third function that returns the states in a specific risk category using the first two functions.

- Assignment 3 – "Analyzing Black Lives Matter Data" assigned in Week 9 – includes one function that reformats a dictionary into two lists based on whether there were BLM protests in that state. The second function returns the average value of a provided field from an inputted list of dictionaries, and the third function uses the two functions to compare the average rates of a provided field between states with and without BLM protests.

The CS1 course used EdStem [8] – a platform that features an online compiler for students to complete programming assignments. After the course, EdStem provided us with every snapshot at run time for each consenting student. Our analysis focused on the three assignments we created, which included 331, 344, and 370 submissions for assignments 1, 2, and 3 respectively. In total, we collected over 52,000 snapshots of student code.

Analysis of Student Code Snapshots: Since the code snapshots represent real-world student programming behavior, we used them to automatically replicate our manual analysis. We detected the change between two sequential snapshots with a Python library called `difflib` (like the `git diff` command between consecutive commits). This library helped us identify when students were adding code, editing code, or testing code. By parsing the code in each snapshot, we extracted features such as the size of chunks students added and the number of debugging steps per added chunk. Finally, we used the EdStem snapshots to evaluate our metric by making manual assessments of the sequences and comparing them to our metric's assessments.

4 RQ1 RESULTS: UNDERSTANDING INCREMENTAL DEVELOPMEHT

We relied on manual observations of the student coding interviews to identify common behaviors in students' programming processes and to develop an understanding of incremental development.

4.1 Key Observations

Categories of Code Changes: Because incremental development broadly entails testing after adding a chunk of code, we wanted to categorize the types of edits to distinguish steps where students added code or debugged code. We found three types of code changes between compilations:

(1) Steps in which students added new code
(2) Steps in which students debug or fix their code
(3) Steps in which students only edit test cases or print statements

By categorizing edits, we created a sequence of labeled steps for each interview where each label represents the student's type of edit between compilations.

Excessive Debugging After Large Additions: We noticed several instances of excessive and ineffective debugging after large chunks of code were added by students. Three of the 15 interviewees wrote nearly the entire function at once, but had to make numerous adjustments to their code to fix syntax errors or logical mistakes. All 3 students needed to use 10 or more debugging steps to fix the issues in the initial large chunk of code. Among these students, we noticed ineffective debugging changes that were either syntactically incorrect or simply unrelated to the error. For example, in one of the many debugging steps after the initial large snippet, Student 5 added a colon to a statement that appended a value to a list, which created an additional syntax error.

Differing Amounts of Struggle: Our observations led us to define the concept of "struggle" during the coding process as the amount of adjustments done by a student to fix any syntactic or semantic errors. We noticed significant variation in students' abilities to solve the tasks, leading to a difference among the students' levels of struggle. In total, 4 of the 15 students wrote nearly the entire function at once with almost no adjustments needed. Further, 7 students finished the task in under 20 minutes, and 6 finished in 5 or fewer compilations. Conversely, 4 took an hour or more to reach a solution, 3 students ran their code at least 20 times in their development process, and 5 students needed prodding towards a solution from the interviewer while stuck in a series of errors. We understood this variation to represent the range of skills and experience among students in a CS1 course.

4.2 Defining Incremental Development

The results from our qualitative analysis helped us answer the series of questions in Section 1 about how much code should be added before testing. Although 3 students wrote the entire function at once and experienced significant struggle when debugging the code, four other students were able to write a similar-sized chunk of code without facing much struggle. Therefore, we reasoned that what constitutes "a small snippet" of code varies based on a student's proficiency, and we can determine whether the snippet was too large for the student based on how many steps it took them to resolve the errors from the code snippet.

In other words, incremental development and struggle are closely linked. Incremental development is a process in which a student adds a *manageable* amount of code in a step, in which a manageable amount is determined by how much the student struggles to fix the chunk of code. As students will need to debug the few syntactic or logical errors that arise, a modest amount of struggle is expected during the programming process. However, we maintain that non-incremental programming behavior is characterized by a significant amount of struggle after a student writes a large chunk of code.

An advantage to this framing of incremental development is that it accounts for the variation in student proficiency that we observed. An issue with using raw wall-clock time or an absolute threshold for a "large chunk" of code is that students will take varying amounts of time to complete a task and may write different-sized chunks based on the complexity of those chunks.

5 RQ2 RESULTS: AUTOMATICALLY MEASURING INCREMENTAL DEVELOPMENT

The results reported in this section explain our process of automatically detecting the key observations from Section 4.1 in the EdStem code snapshot data. To construct a measure of incremental development, we aimed to replicate the manual process of categorizing code changes, observing varying levels of struggle, and penalizing students for excessive debugging after large changes.

5.1 Automatic Detection of Key Observations

Categorizing Code Changes: Using the `difflib` library, we automatically categorized the types of edits into 3 groups, which map to our categories of code changes in Section 4.1.

(1) Steps in which students add new lines of code were classified as *forward progress steps*
(2) Steps in which students only made edits to existing lines of code were classified as *adjustment steps*
(3) Steps in which students only added or edited test cases or print statements were classified as *testing steps*

Quantifying Struggle: To quantify the struggle that students face while programming, we looked for a way to leverage the number of adjustment steps associated with each forward progress step[1]. Here, we generalized the work from Becker [3], who presents a quantification for the amount of repeated errors a student experiences, called the Repeated Error Density (RED). The RED uses the number of consecutive compilations where students get the same error message to quantify the density of that error in a sequence. The metric returns a higher value as a student performs more compilations with the same repeated error. We interpreted the RED as the degree to which a student struggles to fix a specific error.

Similarly, our quantification of struggle per forward progress step captures the degree to which a student struggles to fix any error – syntactic or semantic – that arises from a forward progress step. In this step, we make the assumption that adjustment steps represent attempts to fix issues in a student's code. Equation 1, derived from the RED [3], displays our quantification of struggle for a forward progress step s, where a_s represents the number of adjustment steps needed for that forward progress step:

$$\texttt{struggle(F)} = \frac{(a_F)^2}{(a_F + 1)} \tag{1}$$

Table 1 provides the struggle for varying numbers of adjustment steps per forward progress step. In the "Sequence" column, "F" represents forward progress steps and "A" represents adjustments.

Table 1: Quantified struggle for example sequences

Sequence	Struggle
F ...	0
F A ...	0.5
F A A ...	$1.\overline{3}$
F A A A ...	2.25
F A A A A ...	3.2

An advantage to this sub-metric is that struggle increases as the number of adjustment steps increases. However, we noted before that a small amount of struggle is expected during the programming process. Our quantification of struggle aligns with this idea since the penalty for an adjustment step later in the sequence is greater than an adjustment step at the beginning of the sequence – the penalty for one adjustment step as opposed to zero steps is only 0.5, whereas the penalty for four adjustment steps as opposed to three steps is 0.95.

[1]Note that when categorizing code changes, our algorithm tracks the content of edited lines to link each adjustment step with the one or more forward progress steps that the student was fixing. If a student adjusts code from more than one forward progress step, we count the adjustment step once for each forward progress step that is adjusted.

Penalizing Excessive Debugging After Large Additions: Once we applied the modified RED to quantify struggle per forward progress step, we needed to create a single measure from the collection of struggle values. Recall that a key takeaway from our interviews was that non-incremental programming behavior was characterized by many debugging steps after a large chunk of code. To penalize this behavior in our automated metric, we decided to weight the amount of struggle for a forward progress step based on the size of the step. With this weighting scheme, we penalize sequences of significant struggle after large additions to the code, aligning with our understanding of incremental development in Section 4.2. Moreover, we reasoned that a large addition of code represents a greater proportion of the student's programming process than a small addition of code, so we want these larger additions to contribute to the measure of incremental development more than the smaller additions.

We evaluated several options to quantify the size of code added in a chunk, including the number of lines, the number of lexical tokens, and the Halstead Volume [10] of the chunk. All three options performed the same in our quantitative evaluation that we conducted later (see Section 6). Therefore, we opted to use the number of lexical tokens in a forward progress step to represent size, since line counts can theoretically vary based on programmer style or affordances of different programming languages and Halstead Volume introduces unnecessary complexity for the same result. We present Equation 2 to calculate the proportional size of a forward progress step s, assuming the function `numTokens(s)` returns the number of tokens in forward progress step s:

$$\texttt{size(F)} = \frac{\texttt{numTokens(F)}}{\sum_{F=0}^{N} \texttt{numTokens(F)}} \tag{2}$$

5.2 The Measure of Incremental Development

Our automatic detection of the common incremental development behaviors motivated an automated metric, which we call the "Measure of Incremental Development" (MID).

Input: The input for the MID is a sequence of snapshots collected at compilation time throughout a student's development process for a programming task. The task can be any length and does not need to include multiple functions for our metric to make an assessment.

Algorithm:

(1) Label each snapshot as a forward progress, adjustment, or test code step based on the categories in Section 5.1. We ignore test code steps since these edits do not affect the functionality of the code.
(2) Compute the struggle and size of each forward progress step using Equations 1 and 2.
(3) Calculate a weighted average of the struggle values using the proportional sizes of the forward progress steps as weights.

The MID can be written as follows, where s represents a single forward progress step out of the n total forward progress steps:

$$\texttt{MID} = \sum_{F=1}^{N} \texttt{size(F)} \cdot \texttt{struggle(F)} \tag{3}$$

Figures 1 and 2 show the metric in action. Each white rectangle represents a forward progress step, scaled based on the size of the step in tokens. Each black rectangle following a white rectangle indicates an adjustment step for that forward progress step. To obtain our final measure of incrementality, we compute the average of the values in the "Struggle" row by using the values in the "Proportion of Total Size" row as the weights.

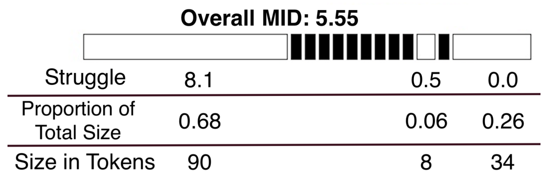

Figure 1: Example of non-incremental development from student code snapshot data

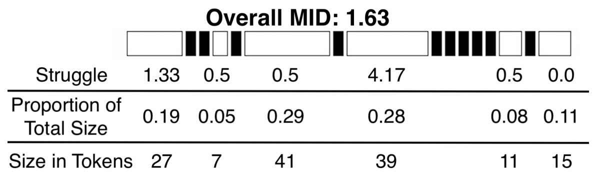

Figure 2: Example of incremental development from student code snapshot data

Output: The MID returns a score greater than or equal to 0. If a student experiences no struggle – indicated by a student using zero adjustment steps – then they will get a score of 0. As the amount of struggle increases, especially after large forward progress steps, the metric score increases. A higher metric score signals an ineffective development process (i.e., non-incremental) because the student likely experienced significant struggle after a larger progress step. Conversely, a lower score indicates that a student likely used an effective incremental development process for their skill level.

Interpreting MID Values: To identify a score threshold between incremental and non-incremental development, two team members independently conducted a review of 40 sequences of snapshots and labeled each as incremental or non-incremental based on manually detecting the common behaviors of non-incremental development identified in Section 4.1. The reviewers agreed on 35 of the 40 sequences, resulting in a Cohen's kappa statistic of 0.68 (substantial agreement) [16]. After deliberating on the 5 disagreements, the reviewers finalized two categories of 28 incremental sequences and 12 non-incremental sequences. We then applied our metric to each of these sequences. By comparing our manual label to the score returned by our metric, we found that a score of 2.5 is the optimal separator between incremental and non-incremental development. Using the threshold of 2.5, our metric correctly labelled 33 of the 40 (82.5%) sequences. We found that a score below 2 typically indicates incremental behavior and a score above 3 typically indicates non-incremental behavior. A score between 2 and 3 consisted of many "close-calls", as labelled by the reviewers, so it was harder to make a decision on these. Table 2 summarizes these classifications.

We have posted our metric publicly as a Python package called `measure-incremental-development` so researchers can apply the MID to their own sequences of snapshots [19].

Table 2: Interpretation of values outputted by MID

Score	Label
0 - 2	Likely incremental
2 - 2.5	Somewhat incremental
2.5 - 3	Somewhat non-incremental
3+	Likely non-incremental

6 EVALUATION

6.1 Theoretical Evaluation

To evaluate our understanding of incremental development against existing interpretations of it, we examined prior work that discusses the history, purpose, and goal of incremental development. In the influential work "Iterative and Incremental Development: A Brief History" by Larman and Basili [15], the authors discuss the shift away from the "waterfall" method, which involves fully developing the project code before testing the entirety of the program. Among the primary reasons for adopting incremental development in the 1960s was that it "leads to a more thorough system shakedown" and "avoids implementer ... discouragement" [9, 15]. Similarly, 1970's software-engineering thought leader Harlan Mills further espoused incremental development, noting that the waterfall method may result in a program that "exceeds our human intellectual capabilities for management and control" [15, 17]. Although these philosophies were written about industry-level programming projects, the idea that incremental development is used to reduce programmer struggle and to keep a program in a manageable state is applicable to student programming in CS courses.

Figures 1 and 2 show how our measurement of incremental development aligns with this understanding of incremental development. The sequence in Figure 1, according to our threshold, is considered non-incremental behavior. The student adds 70% of their code in one step and spends 9 adjustment steps fixing the errors, which is precisely the type of behavior we want to detect as non-incremental. Although the two other forward progress steps are effective, the majority of the program was written non-incrementally, resulting in the high score (MID = 5.55).

On the other hand, the sequence in Figure 2 is labelled as incremental. The student adds code in manageable chunks, indicated by the limited adjustment steps after each forward progress step. Although the student experienced greater struggle during the fourth forward progress step, this was not an especially large chunk of code compared to other chunks, resulting in a low score (MID = 1.63). Interestingly, the sequences in Figures 1 and 2 have 10 total adjustment steps. However, the student in Figure 1 used nearly all the adjustments on the large code addition, so they were penalized much more compared to the student in Figure 2 who used fewer adjustments across smaller-sized additions.

6.2 Quantitative Evaluation

We evaluated our metric against human raters to check if our metric agrees with a manual rating of a sequence of snapshots.

Two human reviewers evaluated 20 pairs of randomly-selected and randomly-paired sequences. On average, each reviewed sequence contained 97 snapshots. The reviewers evaluated a pair

		MID Label		
		Incremental	Not Incremental	Total
Human Label	Incremental	12	2	14
	Not Incremental	1	5	6
	Total	13	7	20

Table 3: Comparison of MID's labels to human reviewer labels

by looking at every snapshot in both sequences to decide which sequence was more incremental. The reviewers kept note of the "close-calls," which indicated tough decisions for the reviewers. The reviewers agreed on 16 of the 20 pairs. The Cohen's kappa value here was 0.56 (moderate agreement) [16]. The reviewers then deliberated to reach a consensus on the 4 pairs with disagreement. We compared our final list of human choices to our metric's choices on the same 20 pairs and found that the metric agreed with humans on 16 (80%) of the 20 pairs. Among the four misclassified pairs, three were marked as close-calls, indicating that the reviewers also had a hard time classifying these pairs.

We also evaluated whether the threshold of 2.5 was an accurate cutoff to separate incremental and non-incremental sessions. Two reviewers labeled a new set of 20 sequences as incremental or not, marking tough decisions as close-calls and resolving any disagreements through deliberation. Our reviewers agreed on 17 of the 20 labels, resulting in a Cohen's kappa of 0.66 (substantial agreement) [16]. We then applied the MID to the 20 sequences, classifying a sequence as incremental if the metric returned a score below 2.5. The metric agreed with our reviewers on 17 (85%) of the 20 sequences, displayed in Table 3. Of the 3 incorrect classifications, 2 of them were classified as close calls and had scores of 2.8 and 3.7. The other incorrect classification was one that reviewers marked as non-incremental, but received a score of 1.3.

7 DISCUSSION

The high accuracy between the manual assessment of incremental development and the automated assessment from the MID show promise that our metric can accurately determine the level of incremental development used in a programming session.

Advantages Compared to Prior Work: Since we evaluated the MID on its ability to replicate manual evaluations of incrementality, our work separates the students' development practices from their outcomes. Previous metrics [3, 6, 13, 14, 25] discussed in Section 2 extract features from a development process to predict outcomes such as submitting on time or passing all test cases. While these metrics are certainly valuable for predicting assignment grades and informing interventions, our metric serves as a way to capture whether the coding process itself is incremental or not.

The MID also offers a unique perspective on incremental development and problem decomposition. Unlike previous approaches such as Charitsis' NLP approach [7], our metric examines each student in the context of their own capabilities, meaning that the ideal size of the decomposed problems varies from student to student.

Further, the MID can make an assessment on relatively easy-to-collect data. The only input is a sequence of snapshots at run-time, which many modern or online IDEs, such as EdStem, can collect; other metrics require data that is more difficult to collect such as compiler output or wall-clock time [3, 6, 14], or they depend on features of a specific IDE [5, 11, 12].

Threats to Validity: A core component of incremental development involves the programmer planning out the stages of their development process. Without knowing the programmer's development plan, we have to make some assumptions about the programmer's intent that threaten our metric's validity.

One assumption of this metric is that a programmer's newly added code corresponds to an attempt at adding behavior. However, this may not be fully representative of the programmer's thought process. For example, a student could make a one-line addition of code with the intent to debug an error. We would count this as forward progress when it truly represents an adjustment. However, since we weight forward progress steps based on their size to penalize large changes, we proceed with this assumption. Nonetheless, alternate solutions that capture this intent may exist and exploring these approaches is certainly an avenue of future work.

We also assume that we can match adjustments to forward progress steps based on locality. We do this because we noticed that programmers may go back to previous additions to make delayed edits, so associating these adjustments with only the most recent forward progress step did not seem to reflect the programmer's intent. However, this assumption does not take into account that adding new code may require the user to edit preexisting code to achieve compatibility. In this case, it is possible that the metric performs poorly on longer and more complex problems.

Potential Applications: Our metric can be immediately used to determine the level of incremental development used by CS1 students. The results of such a study can help researchers identify ways to improve student development processes.

Our metric also enables real-time assessment through an IDE-plugin that automatically computes the MID as students program. With such a tool, educators can promote incremental development among students by monitoring students' programming processes and intervening as necessary. Further, the MID supports future research endeavors by allowing instructors to assess the effect of pedagogical strategies on students' programming behaviors.

8 CONCLUSION

In this work, we identified common behaviors of incremental and non-incremental development among CS1 students. We found ways to automatically detect these behaviors in a sequence of coding snapshots at compilation, culminating into a formal metric – the Measure of Incremental Development (MID). The MID aligns with prior work that frames incremental development as a process that keeps a program in a manageable state and automatically evaluates students' development with over 80% accuracy. Therefore, our work makes an important contribution to the computing education community: an accurate and automated metric to measure a student's adherence to incremental development.

ACKNOWLEDGEMENTS

This work was supported in part by NSF IUSE award 2044473.

REFERENCES

[1] Lewis Baumstark and Michael Orsega. 2016. Quantifying Introductory CS Students' Iterative Software Process by Mining Version Control System Repositories. *J. Comput. Sci. Coll.* 31, 6 (jun 2016), 97–104. https://doi.org/10.5555/2904446.2904470

[2] Kent Beck and Erich Gamma. 2000. *Test-Infected: Programmers Love Writing Tests*. Cambridge University Press, USA, 357–376.

[3] Brett A. Becker. 2016. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 296–301. https://doi.org/10.1145/2899415.2899463

[4] J. Calderhead. 1981. Stimulated Recall: A Method for Research on Teaching. *British Journal of Educational Psychology* 51, 2 (1981), 211–217. https://doi.org/10.1111/j.2044-8279.1981.tb02474.x

[5] Adam S. Carter. 2012. Supporting the Virtual Design Studio through Social Programming Environments. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 157–158. https://doi.org/10.1145/2361276.2361309

[6] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/2787622.2787710

[7] 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* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 113–120. https://doi.org/10.1145/3491140.3528272

[8] Edstem. 2022. *Edstem*. https://edstem.org/

[9] R. L. Glass. 1969. An Elementary Discussion of Compiler/Interpreter Writing. *ACM Comput. Surv.* 1, 1 (mar 1969), 55–77. https://doi.org/10.1145/356540.356545

[10] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software Complexity Analysis Using Halstead Metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. 1109–1113. https://doi.org/10.1109/ICOEI.2017.8300883

[11] Matthew C. Jadud. 2006. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research* (Canterbury, United Kingdom) *(ICER '06)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1151588.1151600

[12] Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer. 2017. DevEventTracker: Tracking Development Events to Assess Incremental Development and Procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 104–109. https://doi.org/10.1145/3059009.3059050

[13] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2017. Quantifying Incremental Development Practices and Their Relationship to Procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 191–199. https://doi.org/10.1145/3105726.3106180

[14] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 407–413. https://doi.org/10.1145/3287324.3287366

[15] Craig Larman and Victor R. Basili. 2003. Iterative and Incremental Developments: A Brief History. *Computer* 36 (2003), 47–56.

[16] Mary L. McHugh. 2012. Interrater Reliability: The Kappa Statistic. *Biochemia Medica* 22 (2012), 276 – 282.

[17] H. D. Mills. 1976. Software Development. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 388. https://doi.org/10.5555/800253.807706

[18] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

[19] Anshul Shah. 2022. measure-incremental-development. https://pypi.org/project/measure-incremental-development/

[20] Anshul Shah. 2022. Programming Tasks to Observe Incremental Development. https://bit.ly/program-tasks

[21] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

[22] E. Soloway and James C. Spohrer. 1988. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., USA. https://doi.org/10.4324/9781315808321

[23] CORPORATE The Joint Task Force on Computing Curricula. 2001. Computing Curricula 2001. *J. Educ. Resour. Comput.* 1, 3es (sep 2001), 1–es. https://doi.org/10.1145/384274.384275

[24] Maureen M. Villamor. 2020. A Review on Process-oriented Approaches for Analyzing Novice Solutions to Programming Problems. *Research and Practice in Technology Enhanced Learning* 15, 1 (Apr 2020), 8. https://doi.org/10.1186/s41039-020-00130-y

[25] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. 2013. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. 319–323. https://doi.org/10.1109/ICALT.2013.99