A Process for Collecting and Analyzing Chronological Data for CS1 Programming Assignments

Raymond Pettit[1], Ryan Clements[1], Susan Mengel[2]
[1]School of IT and Computing, Abilene Christian University, Abilene, TX, USA
[2]Computer Science, Texas Tech University, Lubbock, TX, USA

Abstract - *In this paper, the use of chronological data is explored in analyzing students' submitted programming assignments. By capturing intermediate versions of a student's program during development, the progression of steps can be seen that the student took in attempting to solve the problem. This chronological data can be used to give instructors additional meaningful information as to a student's understanding of programming concepts and yet does not add any additional burden to the student while completing the assignment.*

Keywords: CS1, program assignments, computational thinking, assessment

1 Introduction

It is a challenge to access the mastery of concepts in college-level entry programming courses. Students come to a CS1 course from a diverse set of backgrounds. Some students have had much exposure to programming in high school, while others have had none.

The failure rate in college-level introductory computer science courses may be 30% or higher [1][2]. This failure rate found in a college's CS1 course may be one of the higher failure rates across campus. One of the contributing factors is that it is often difficult for an instructor to make an accurate assessment of what a student really understands at any given point in the course [5].

As programming skills are usually a goal of a CS1 course, instructors typically assign work in the form of programming assignments in which students must write a small, but fully functioning program that meets certain requirements. For a thorough discussion of ways to assess mastery of CS1 concepts, see [6].

For instructors, often only the end product of the student's efforts is examined; i.e., the completed program. From this completed program, correctly working or not, an assessment of the student's mastery over the given material is made.

One problem with only looking at the end product is missing the student's journey to get to the end, which can show quite a bit about the student's level of understanding, especially with regards to higher-level concepts and problem solving. An instructor who observed a student throughout the assignment would have more information from which to make an assessment of that student's mastery.

In order to capture the data of the student's journey to complete a programming assignment, the authors propose a process described in this paper as an experiment to capture some of that data and to see if it could be beneficial in assessing students. If the process is proved feasible and beneficial, it could be incorporated into many assignments in the future.

2 Background

In the CS1 course in which the experiment took place, the programming assignments were divided into two groups, in-class assignments and out-of-class assignments. Control could not be exercised over the resources that students used to complete an out-of-class assignment. Resource availability for in-class assignments, however, was strictly controlled. The in-class programming assignments consisted of both the non-computer-based write-out-the-program-on-paper style and the computer-based write-and-run assignments. Both of these styles were used for exams, quizzes, and in-class practice.

Part of the motivation of this work besides looking more into the student's problem solving process is to begin to develop a set of instruments to measure the student's ability to perform computational thinking [7] as introduced by Jeannette Wing. She defined computational thinking as embodying the skills of abstraction and automation. Subsequent work [8] has expanded upon the definition to include algorithms and design. To gain insight into the students algorithmic and design processes would be facilitated by observing how they solve problems.

Another part of the motivation of this work comes from observing students in lab who haven't truly mastered the material, but who can often produce a correctly working program, either through sheer luck or trying a multitude of approaches until something works. It is because of these motivations that the authors do not want to rely entirely on the finished product (the correctly working competed program) to assess the skill of the student.

3 Related Work

Work has been done to evaluate student progress within a single programming assignment by capturing and examining each compilation attempt [3]. Information captured this way can aid in analyzing the most difficult sections of the program and also point to areas of frustration. The authors automated the process of examining compiled submissions, but needed a substantial amount of training data for reasonable accuracy. Data examined included average number of consecutive compilations with the same edit location, average number of consecutive pairs with the same error, the average time between compilations, and average number of errors.

Intention-based scoring is a similar strategy that attempts to use students' intermediate submitted work (in the form of compile attempts) [4]. The goal of the process is to assess how close a student's initial attempt is to the correct solution. Examination of the intermediate forms was done manually and resulted in the classification of the bugs found.

4 Methodology

4.1 Participants

This experiment was performed during the Fall 2011 and Spring 2012 semesters in a required class, CS115: Introduction to Programming Using Scripting, for students in the following majors: Information Technology, Information Systems, Digital Entertainment Technology, and Math Education. The class consisted primarily of students from one of these four majors. The programming language used in this course was Python 3. No prior experience in programming was required to enroll in this course.

In the Fall 2012 semester, 26 students were given the assignment. There were 23 students who were given this assignment in the Spring 2012 semester. In both cases, this assignment was given as an in-class quiz which had a small (1%-2%) impact on the student's final grade. All students who were in attendance on the scheduled day took part in the assignment as a standard required quiz.

4.2 Procedure

Several methods for collecting chronological data during a programming assignment were explored. One method was to create a custom editor that the students would be required to use while completing the assignment. The custom editor would store data as the user entered it. This custom editor would give a great deal of flexibility and power to record all of the data that was wanted, but would require a significant amount of effort to build and deploy.

Another possibility was to do direct keystroke logging during the quiz period. The data could then be reconstructed and analyzed later offline. While keystroke logging programs are available, there would have been a significant amount of manual effort in implementation and processing the data received.

Fortunately, it was not necessary to capture every keystroke to obtain the information that was desired. The main concern was with the ordering of steps while completing the assignment. All that was needed was a lower resolution "playback" of the programming session. To this end, the authors experimented with the built-in archiving capabilities of the Google Documents editor. The Google Documents editor saves a new copy of a file after any 10 seconds of inactivity. Each of these intermediate versions can be easily accessed at a later time. By using Google Documents, the authors would also have the ability to collect assignments easily from the students when they finished and be able to store a permanent archive that could be accessed in the future.

In order to prepare for the experiment, a shared Google Document was created for each student. As the campus where the trial took place is a Google Apps Campus, all student user accounts are already connected to Google Documents. Creating a shared Google Document for each student, therefore, was a straightforward effort and took about one minute per student to accomplish. The authors are exploring ways to make shared document creation more of an automated process in the future. Even if students do not already have a Google account, they can create one for free.

On the day of the quiz, students simply looked at their Google Documents folder and found the shared document bearing the chosen name (i.e., CS115.01_Quiz_4.txt). This document opened as an empty document within Google's web-based text editor. Students were then given the problem description and directed to use the editor to write the solution.

This assignment was very similar to a paper and pencil- based programming test in that students did not have access to an integrated development environment (IDE), compiler, or interpreter. They were not allowed to do trial runs of the program and iterate on the feedback given in order to capture a more accurate picture of what the student fully understood.

The only benefit provided by the editor was the ability to manipulate text in standard ways such as deleting or moving previously typed lines. Although students can delete previously input text, the deleted versions are still saved as part of the Google Documents archiving process.

4.3 Data Collection

Data was collected in the form of intermediate versions of the modified text file. A researcher then stepped through the intermediate forms of a student's completed coding assignment while recording the order in which designated milestones were accomplished. Within Google Documents, a document owner can choose to view a detailed revision history of a given file. Once the beginning version of the file is selected, then it is a simple matter of stepping through the various revisions until the final version is reached. For example, typically one to two lines of new code per revision was seen.

Before the submitted programs were opened, the researcher analyzed the assignment and designated certain required milestones that must be present in a correctly working program. Milestones included: collecting input, printing output, declaring a function, calling a function, and performing a calculation.

CS115, Quiz 4, Castle Defense

You're in charge of the castle's perimeter defenses. Your first round of defense is catapults, but they will only kill a certain percentage of the enemies. After the enemies get past the catapults, your second line of defense is archers, but they will only kill a certain percentage of the enemies that had gotten past the catapults.

Write a program that will help you to predict the number of enemies that will get past both defenses (so you can plan for the boiling oil). The program should ask the user for the number of enemies, the expected catapult kill percentage, and the expected archer kill percentage.

Print out the total number of enemies expected to make it past both defenses and through to the castle (round the answer to the closest whole number). For values of 1000 or more, insert commas in your output.
Ex. 1,000 or 15,324 etc.

Define a main function exactly as shown here- **def main():**
From within the main function, ask for data from the user and do the printing. There should not be any user input or printing in any function besides **main().** Do not use global variables in the program. Any call to the **left_standing** function should be done from within the **main** function.

Create a function that, given the number of people in a group and a kill rate, will return the number of people that survived. Include all mathematical calculations in this function. Do not do any mathematical calculations outside of this function. Define this function exactly as shown here- **def left_standing(group, kill_rate):**

Sample Runs

```
Castle defense
Enter number of enemies: 100
Enter catapult kill percentage: 40
Enter archer kill percentage: 50
Total enemies remaining: 30
```

```
Castle defense
Enter number of enemies: 10
Enter catapult kill percentage: 50
Enter archer kill percentage: 50
Total enemies remaining: 2
```

```
Castle defense
Enter number of enemies: 12345
Enter catapult kill percentage: 10
Enter archer kill percentage: 20
Total enemies remaining: 8,888
```

Figure 1: The Castle Defense Assignment

For the assignment later described in this paper (about 15 lines of code and 8 milestones), it took a student researcher about one minute to step through a single student's assignment and record the ordering of the milestones. In the future, it may be possible to automate this process by specifying variable and/or function names and using a program that analyzes text.

At the end of this data collection process, a spreadsheet containing a milestone ordering for each student is filled out. From there, the data can be analyzed in a variety of ways as given below.

5 The Sample Problem

The process described in this paper has been used for two semesters in a CS1 course at Abilene Christian University. The first assignment that was given in this way is titled "Castle Defense." This assignment was given in the Fall 2011 semester and the Spring 2012 semester. The Castle Defense assignment as it was given to the students is shown in Figure 1.

This assignment was created to test understanding of functions and their use. By restricting the student to only use the function definitions given in the problem statement, the student must see the calculation required as composed of two similar calculations that can be performed to get the correct answer. Many students only see functions as a means of taking code from one part of a program and putting it in another section. They may have difficulty in understanding the value of having a function which can receive values, process them, and return a resultant value. They would not initially see a function as something that could be called multiple times. It is because of this misperception of functions that this assignment is of value in helping instructors understand deficiencies in students' perceptions of functions.

In order to assess a student's completed assignment as objectively as possible, a rubric was created for grading the assignment. This rubric was used strictly to analyze the final version of the program that the students completed. Each student earned weighted points for each of the 22 conditions that were accomplished successfully. The rubric is shown in Table 1.

In order to record the chronological data for this assignment, milestone markers were created. These milestones are required accomplishments along the way to completing the program correctly. However, these milestones could be achieved in any order and still produce a correct end product. It is through this analysis of the ordering that instructors can gain insights into the level of conceptual understanding of a student and their ability to design. These milestones are:

1.) Student opened file
2.) Function main() is defined
3.) Function left_standing() is defined
4.) Function left_standing() is called the 1st time
5.) Function left_standing() is called the 2nd time
6.) Input is requested of the user
7.) Print statement shows final result
8.) Body of left_standing() started

Table 1: Grading Rubric

	Castle Defense Grading Criteria CS115.01 Quiz 4	weighting
1.)	Both functions declared as stated	4
2.)	Mathematical calculations are only in left_standing()	4
3.)	Input calls are only done from main() function	4
4.)	Ouput is only done from within main()	4
5.)	left_standing() is not called outside of main()	4
6.)	main() is called	2
7.)	Proper output for header text ('Castle defense')	2
8.)	Values returned from input statements are stored or used	6
9.)	Input values are converted to number types	5
10.)	An attempt is made to print the final result	5
11.)	The final result is formated properly (text and commas)	3
12.)	left_standing is called at least once	7
13.)	left_standing() is called properly the first time	4
14.)	The value returned from the first call to left_standing() is stored or used	7
15.)	left_standing is called twice	7
16.)	left_standing() is called properly the second time	4
17.)	The value returned from the second call to left_standing() is stored or printed	7
18.)	left_standing() contains correct mathematical calculation	4
19.)	left_standing() returns a calculated value	5
20.)	All referenced variables are in scope (not including undeclared variables) (B)	5
21.)	All variables used have been declared (B)	5
22.)	All variables declared are used (B)	2
	Total	100

The first milestone simply gives a way to record the time at which the programmer opened the file for editing. Including 1) as a milestone allows for keeping elapsed times between milestones. The second milestone is fulfilled when the programmer completes the official function definition for the main() function. The third milestone is fulfilled when the left_standing() function is officially defined. In order to complete this programming quiz successfully, the left_standing() function must be called twice. Milestones 4 and 5 are fulfilled when these functions are officially called. Milestone 6 is fulfilled when the programmer uses the input() function to request input of the user. Milestone 7 is fulfilled when the program attempts to print out the final answer. The result may not be calculated yet, but the call to print the answer is made. The final milestone, 8, is fulfilled when the body of the left_standing() function has been started. The body included any work on the left_standing() function besides its definition or a blank return statement.

6 Results

6.1 Summary Data

For each student, an analysis was made as to the order in which the milestones were accomplished. These individual orderings were summed and the average taken. The summary information for the assignment given in the Fall 2011 semester and the Spring 2012 semester is shown in Tables 2 and 3.

The ordering summary data indicates that the two different groups of students, on average, completed the problem in similar progressions. As well, the standard deviation for a given milestone is not significantly different from one semester to the next.

Elapsed time ranged from a few minutes to a little over 10 minutes on each milestone. Grades ranged from as low as 20 to as high as 100 using the grading rubric in Table 2.

Table 2: Ordering Summary for Fall 2011 (n=26)

Milestone	Average	Standard Deviation
Student Opened File	1.00	0.000
main() defined	2.88	0.927
left_standing() defined	3.22	1.808
left_standing() 1st call	5.85	0.988
left_standing() 2nd call	7.08	0.793
Input gathered	3.80	0.957
Print final result	5.40	1.414
Body of left_standing() started	6.05	1.588

Table 3: Ordering Summary for Spring 2012 (n=23)

Milestone	Average	Standard Deviation
Student Opened File	1.00	0.000
main() defined	2.87	1.014
left_standing() defined	3.57	1.619
left_standing() 1st call	6.10	1.044
left_standing() 2nd call	7.36	0.809
Input gathered	3.74	1.096
Print final result	5.56	1.562
Body of left_standing() started	5.36	1.590

6.2 Comparison to Expected Orderings

There are many different orderings of these milestones that show a logical progression through the program. The authors created two straightforward orderings, to see if these were indeed the most popular. The first ordering suggestion is based on a novice following the flow of control of a program and basing their ordering of code completion on the program's flow. The second suggestion is based on what the more experienced programmers indicated as their preferred progression through the code. This ordering involved creating and finishing the main function, including calling the auxiliary function, before creating the auxiliary function. These two orderings are shown in Table 4.

Table 4: Suggested Orderings

Milestone	Flow of Control	Experienced
Student Opened File	1	1
main() defined	2	2
left_standing() defined	5	7
left_standing() 1st call	4	4
left_standing() 2nd call	7	5
Input gathered	3	3
Print final result	8	6
Body of left_standing() started	6	8

The summary results show that after defining the main() function, the next step (on average) was to define the left_standing() function. This is likely due to the student's beginning the assignment by including the problem constraints first. As the ordering of the defining of the left_standing() function may not be an indicator of the student's thought process, the authors later added the eighth milestone which better records when a student was giving their attention to the left_standing() function.

6.3 Exploring One Student’s Data

As the authors began analyzing the collected ordering data and comparing it to grades earned in the assignment, a few surprises were uncovered. One of these surprises involved a student that scored a much higher grade on this assignment than his course average would have predicted. The student in question had a course grade that placed him as the 19th out of 23 students in the course. The instructor’s subjective assessment of this student’s performance concurred with this student’s low ranking. The surprise was that this student was one of only eight students to score a grade of ‘A’ on this assignment (according to the grading rubric shown in Table 1). The ordering for this student’s assignment is shown in Table 5.

This ordering reflects that the student had not fully thought out his approach to the problem beforehand as main was not completely defined before work began on the left_standing() function. It also raises questions as to his overall approach since he went back and forth between main and the left_standing() function. When the particular student learned of his ‘A’ grade on this assignment, he expressed surprise and stated that he was very unsure of how to create a program to achieve the desired goal.

Table 5: An Individual Student’s Curious Orderings

Milestone	Flow of Control
Student Opened File	1
main() defined	2
left_standing() defined	4
left_standing() 1st call	7
left_standing() 2nd call	8
Input gathered	3
Print final result	6
Body of left_standing() started	5

Additional analysis showed that 16% of the students in Fall 2011 and 21% of the students in Spring 2012 addressed the body of the function left_standing() at steps four or below. The average grade of the Fall students was 61 and the average grade of the Spring students was 58.2 out of 100. In Fall, 60% of the students addressed the body of the function at steps 6 to 8, and in Spring, 43%. The average grade was 79 in the Fall and 78 in the Spring for those students. Clearly, students who follow the ordering suggested in Table 4 tend to do better than those who might follow a different ordering.

There are other interesting cases that have given additional insight into the reasoning of the students as they completed this assignment. The authors would like to create some template ordering patterns that reflect a good approach to this assignment and a poor approach to this assignment. If accomplished, the instructor could receive an automatically generated assessment of the likelihood that a student showed mastery of the concepts in question. This template could be used by the instructor to look deeper into the work of certain students and help identify deficiencies in learning, such as using a more random pattern of orderings in accomplishing assignments and insufficient numbers of students able to complete portions of an assignment.

7 Future Work

Outlined in this paper is an early implementation of a technique for recording additional data that students produce while completing a programming assignment. Currently, it can be a time intensive process to set up an assignment for capture using Google Docs, especially with large numbers of students. Additional work needs to be done to have a more automated means of setting up and collecting data from larger numbers of students.

Analysis of the data is currently done manually and requires a researcher to step though the intermediate versions of a student’s code. The process could be improved by feeding these intermediate versions into a program that could assign the orderings automatically.

Work needs to be done to produce template patterns for more or less desirable orderings. To create templates, several assignments will need to be given to students and a comparison of orderings of experienced to novice students made to gain more insight into how orderings should be done. Certainly, much care must be taken when creating the assignment and selecting the milestones as there likely will be just a small subset of the milestones that are of interest in testing the mastery of a given concept.

8 References

[1] Theresa Beaubouef and John Mason. 2005. “Why the high attrition rate for computer science students: some thoughts and observations”. *SIGCSE Bulletin* 37, 2 (June 2005), 103-106.

[2] Dawn McKinney and Leo F. Denton. 2004. “Houston, we have a problem: there's a leak in the CS1 affective oxygen tank”. *SIGCSE Bulletin.* 36, 1 (March 2004), 236-239.

[3] Ma. Mercedes, T. Rodrigo, and Ryan S.J.D. Baker. 2009. “Coarse-grained detection of student frustration in an introductory programming course”. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (ICER '09). ACM, New York, NY, USA, 75-80.

[4] H. Chad Lane and Kurt VanLehn. 2005. “Intention-based scoring: an approach to measuring success at solving the composition problem”. *SIGCSE Bulletin* 37, 1 (February 2005), 373-377.

[5] João Paulo Barros. 2010. “Assessment and grading for CS1: towards a complete toolbox of criteria and techniques. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli Calling '10). ACM, New York, NY, USA, 106-111.

[6] Anne Venables, Grace Tan, and Raymond Lister. 2009. “A closer look at tracing, explaining and code writing skills in the novice programmer”. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (ICER '09). ACM, New York, NY, USA, 117-128.

[7] Jeannette Wing. 2008. “Computational thinking and thinking about computing”. *Philosophical Transactions of the Royal Society*, 366, 3717-3725.

[8] Ljubomir Perković, Amber Settle, Sungsoon Hwang, and Joshua Jones. 2010. “A Framework for Computational Thinking Across the Curriculum”. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2010). ACM, New York, NY, USA, 123-127.