Analyzing Novice Debugging Behavior Using Programming Process Data

Archan Das

CMU-CS-25-124

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Professor Mark Stehlik (Co-Advisor)
Professor David Kosbie (Co-Advisor)
Professor Roy Maxion (Subject Matter Expert)

Submitted in partial fulfillment of the requirements for the degree of Master of Science of Computer Science.

Keywords: CS Education, debugging, software engineering, process data

This thesis is dedicated to my family, who supported me all the way.

Abstract

Background. Debugging is an important part of the software development workflow. In order to improve the techniques and instruction of debugging, we need to understand the cognitive process through which programmers debug. Previous research has used a variety of methodologies for studying the debugging process, including concurrent verbal protocols, quantitative analyses, and neural imaging. This research has established the sequence of cognitive phases that programmers progress through while debugging. One frontier in this research is the use of process data to study debugging. This process data consists of logs collected from integrated development environments (IDEs) that record the process by which programmers work on code.

Aim. We aim to: a) create a framework for analyzing process data captured from an IDE, b) analyze process data collected from a population of introductory programming students to observe patterns in student debugging behavior, and c) use the collected data to identify efficient and inefficient habits exhibited by students while debugging.

Data. We collected process data across three exercises from 315 students in an introductory programming class. This data consists of an event log of every keystroke, code execution, and submission attempt students made while working on their exercises.

Methods. We extracted a timeline of cognitive phases from the process data for each student and validated our model with a panel of experts. We tested the effect of three behavioral features (use of print statements, time in locate-error phase, and functional edits per cycle) against a novel measure of student efficiency in debugging (count of run-program events), but found results to be inconclusive. We also observed patterns across the subject population of our extracted cognitive phases.

Results. We found that the frequency of print statements had a positive correlation with debugging struggle across all exercises. Increased time spent in locate-error phase had a statistically significant impact on student debugging struggle in some exercises, but not others. Subjects tended to perform faster and more focused repairs to their code later in debugging episodes. Finally, debugging struggle had a weak negative correlation with average exam scores in the course.

Conclusion. Results suggest that students should be encouraged to spend more time reasoning about their code while debugging. Process data also shows promise as a tool for evaluating and giving feedback on the student debugging process at scale. In addition, our framework can be widely useful for experiments on student debugging behavior, especially when large subject populations make alternative methods difficult.

Acknowledgments

This research would not have been possible without a lot of very helpful people. More thanks than I could possibly write goes out to my advisors, Professors **Roy Maxion** and **Mark Stehlik**, for guiding me for the past year. It's been a really positive experience and both of you have taught me a lot about research and life.

I am deeply grateful for Professors **David Kosbie** and **Mike Taylor**, for supporting my work from the 15-112 side of things. The fact that you were willing to dedicate your time and attention to this project really means a lot.

I owe a debt of gratitude to the entire CMU CS Academy team for building a spectacular platform that enabled this whole project. **Austin Schick** and **Evan Mallory** deserve a medal for the great development work they did in support of this research. You guys rock!

More thanks goes out to the panel of experts: **Ethan Kwong**, **Audrey Hasson**, **Elena Li**, **Felicia Zhang**, and Professor **Kelly Rivers**.

And last but not least, a heartfelt thanks goes out to the students of 15-112.

Contents

1 Introduction **1**
- 1.1 Background and Related Work . . . 1
  - 1.1.1 Theoretical Foundations of Debugging . . . 1
  - 1.1.2 Empirical Investigations of Debugging . . . 4
  - 1.1.3 Research focused on novice debugging . . . 4
  - 1.1.4 Differences Between Experts and Novices . . . 5
- 1.2 Cognitive model used in this research . . . 5
  - 1.2.1 Cognitive phases of debugging . . . 6
- 1.3 Gap analysis . . . 7

2 Problem and approach **9**

3 Methods **11**
- 3.1 Apparatus and instrumentation . . . 11
  - 3.1.1 Anonymization . . . 11
  - 3.1.2 CMU CS Academy . . . 11
  - 3.1.3 Process data . . . 12
  - 3.1.4 Debugging exercises . . . 13
- 3.2 Subjects . . . 14
  - 3.2.1 Instructions to subjects . . . 14
- 3.3 Data . . . 15
  - 3.3.1 Intermediate code states . . . 15
  - 3.3.2 Token-level changes . . . 15
  - 3.3.3 Functional and print changes . . . 17
  - 3.3.4 Functional and non-functional execution . . . 17
  - 3.3.5 Extracted event log . . . 18

4 Extracting Cognitive Phases from Process Data **21**
- 4.1 Approach . . . 21
- 4.2 Panel of Experts . . . 22
- 4.3 Participants . . . 22
- 4.4 Materials . . . 23
- 4.5 Procedure . . . 23
- 4.6 Analysis . . . 23
- 4.7 Results . . . 24

4.7.1 Agreement with Automated Labeling . . . 24
4.7.2 Inter-rater Agreement . . . 24
4.7.3 Qualitative Feedback . . . 24
4.8 Cognitive phase log . . . 24
4.9 Discussion . . . 25
4.10 Conclusion . . . 26

5 Analysis of Patterns in Process Data **27**
5.1 Case studies . . . 27
5.1.1 Student 1 . . . 27
5.1.2 Student 2 . . . 28
5.2 Debugging cycles . . . 29
5.3 Count of debugging cycles across exercises . . . 30
5.4 Time evolution of debugging cycles . . . 31
5.5 Correlation with exam scores . . . 32
5.6 Completion grading . . . 33
5.7 The problem with time-elapsed . . . 34
5.7.1 Patterns in time-elapsed across phases . . . 35
5.7.2 The importance of locate-error phase . . . 37

6 Identifying efficient and inefficient behaviors **39**
6.1 Behavioral features . . . 39
6.1.1 Debugging cycles . . . 39
6.1.2 Time spent in locate-error phase . . . 40
6.1.3 Print count . . . 40
6.1.4 Functional edits per repair-error phase . . . 40
6.1.5 Summary of Behavioral Features . . . 41
6.2 Method of Analysis . . . 41
6.2.1 Splitting Groups . . . 41
6.2.2 Statistical Testing . . . 41
6.3 Results . . . 41
6.4 Discussion . . . 42

7 Conclusion **43**
7.1 Summary . . . 43
7.2 Future work - extensions . . . 43
7.3 Future work - applications . . . 44
7.3.1 Educational interventions . . . 44
7.3.2 Intelligent tutoring systems . . . 44
7.3.3 AI debuggers . . . 44

A Exercises **47**
A.1 bowlingScore . . . 47

List of Figures

1.1 **Debugging model** . . . 2
1.2 **Alternative debugging model** . . . 3
1.3 **Cognitive model of the debugging process** . . . 6

3.1 **Example view of a CMU CS Academy exercise.** . . . 12
3.2 **Example keystroke data snippet logged by the learning platform.** . . . 13

4.1 **Cognitive model of the debugging process** . . . 21
4.2 **Extracted cognitive phase timeline for a student submission to the bowlingScore exercise.** . . . 25

5.1 **Extracted event log for student 1.** . . . 28
5.2 **Extracted event log for student 2.** . . . 28
5.3 **Debugging loop.** . . . 29
5.4 **Histogram of number of debugging cycles used by each student in bowlingScore and integerLetterFrequencies exercises** . . . 30
5.5 **Scatter plot of debugging cycles used in bowlingScore and integerLetterFrequencies for students who did both** . . . 31
5.6 **Number of keystrokes and time elapsed in seconds for each repair-error phase, for the integerLetterFrequencies exercise.** . . . 32
5.7 **Correlation between debugging cycle count on integerLetterFrequencies and average exam grades.** . . . 33
5.8 Left: **histogram of grades received by students on the integerLetterFrequencies debugging exercise.** Right: **histogram of debugging cycles used to solve the exercise.** . . . 34
5.9 **Total time elapsed and total time elapsed in locate-error phase (integerLetterFrequencies)** . . . 35
5.10 **Total time spent vs time spent in locate-error phase (integerLetterFrequencies)** 36
5.11 **Percentage of time-elapsed in each cognitive phase (integerLetterFrequencies)** 36

6.1 **The debugging loop.** . . . 40

A.1 **Buggy starter code for bowlingScore exercise.** . . . 49

List of Tables

3.1 **Summary of debugging exercises** 14

4.1 **Agreement between expert and automated cognitive phase labeling.** 24

6.1 **Debugging behavior differences between low-struggle and high-struggle students across two programming exercises.** 42

Chapter 1

Introduction

Debugging is fundamental to software development and has been defined as the process of identifying and correcting errors within a program [Myers, 1978]. Given that debugging may account for up to 50% of software development time [Timmerman et al., 1993], understanding how programmers go about the task of debugging is important. These findings can help inform and improve programming instruction, software engineering best practices, and automated debugging tools.

Over the years, researchers have sought to formalize the process of debugging. These formal models can help programmers, educators, and developers of automated debugging tools. Some researchers have proposed models based on qualitative data, such as verbal protocols, while others have used quantitative data, such as from neural imaging and integrated development environment (IDE) logs.

1.1 Background and Related Work

1.1.1 Theoretical Foundations of Debugging

Theoretical models of debugging describe the steps a programmer performs in order to find and fix bugs. The models also characterize the tactics used by programmers to achieve these steps.

Early Models and Verbal Protocol Studies

Previous research focused on the cognitive processes underlying debugging has used verbal protocol studies (also called think-aloud studies). In these experiments, researchers ask subjects to verbalize their thought process out loud, and analyze the transcripts of each subject's verbalized thoughts. Subjects may be prompted with questions from a researcher while debugging in order to further explore their thought process.

[Katz and Anderson, 1987] conducted one of the earliest studies using verbal protocols to analyze debugging. Their experiments revealed that programmers tend to employ three general strategies to locate bugs:

- Simple mapping from output
- Hand simulation
- Causal reasoning

Their findings suggested that students were more efficient at debugging their own code, compared to fixing code originally written by someone else. The method used to locate the bug did not necessarily affect the success rate of bug repair - once a bug was found in the code, it did not matter how the student found it. Students who were able to find the bugs in their program were usually able to fix them, suggesting that the main difficulty of debugging lies in locating the bug. The basis of this work was the general cognitive model of debugging shown in Figure 1.1.

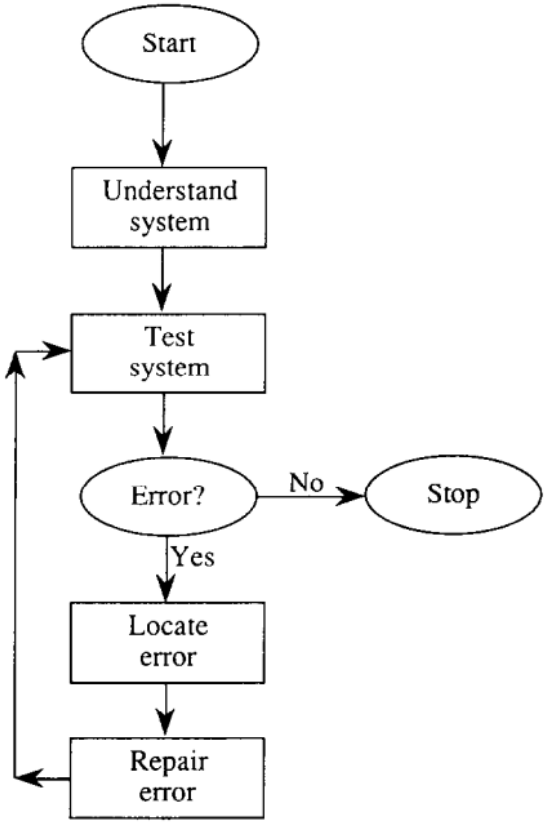

Figure 1.1: **Debugging model**
Each node is one step of the debugging process.

Other verbal protocol studies of the debugging process include [Vessey, 1984], [Vessey, 1985], [Rasmussen and Jensen, 1974], [Whalley et al., 2023], and [Allwood and Björhag, 1990].

An important note to make on verbal protocol studies is that they analyze subjects' thoughts, not their actions. Researchers collect data on what subjects are thinking about while debugging, but generally do not collect data on exactly what subjects are typing into their code editor after doing so.

Verbal protocols have a disadvantage of being resource intensive - analyzing raw transcripts of subjects' thoughts takes a significant amount of time. [Hughes and Parkes, 2003] found that most verbal protocol studies in software engineering involved fewer than 30 subjects. Conducting a high-quality verbal protocol study over a subject population of hundreds is very difficult.

Subsequent Models

Other researchers extended or refined these early models. For example:

- [Vessey, 1985, Vessey, 1984, Vessey, 1986] used a five-phase model consisting of: (1) problem determination, (2) gaining familiarity with the program, (3) exploration of particular aspects, (4) hypothesis formulation, and (5) error repair.
- [Kessler and Anderson, 1986] offered a similar four-phase model: (1) comprehension, (2) detection, (3) localization, and (4) repair.

These cognitive models, while different in exact form, are all similar in how they split the process of debugging. These cognitive models also establish comprehension of the program as a cognitive phase that occurs before programmers attempt to locate errors.

[Kessler and Anderson, 1986] and [Katz and Anderson, 1987] noted that the comprehension phase of debugging is particularly important when subjects are debugging code from a different author. Subjects debugging another person's code had to invest more effort into comprehending that code than subjects who debugged their own code.

Alternative Perspectives

Some scholars have questioned the linearity of these cognitive models. For example, [Gilmore, 1991] argued that the interplay between comprehension and debugging is cyclical rather than strictly sequential. Their model (see Figure 1.2) suggests that understanding the differences between the buggy and the correct version of a program may require multiple iterative cycles.

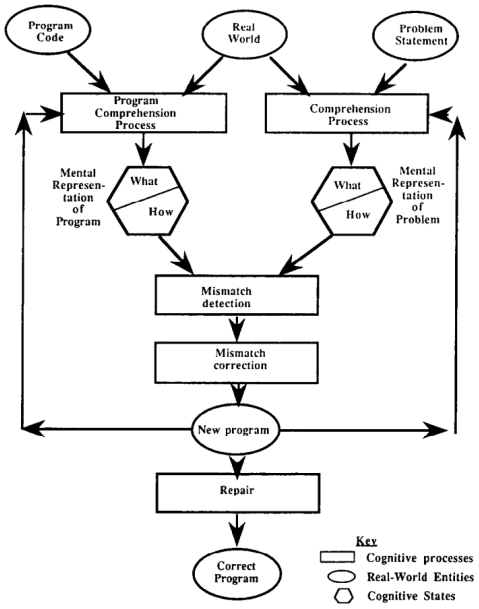

Figure 1.2: **Alternative debugging model**

More recent research [Hu et al., 2024] has suggested that the older models reflect the cognition of programmers while debugging with respect to neural imaging. Thus, we rely on the simpler but seemingly robust models of debugging established by [Katz and Anderson, 1987], [Vessey, 1984], and others as our foundational understanding of the debugging process.

1.1.2 Empirical Investigations of Debugging

Empirical research on debugging has used both qualitative and quantitative methods. Qualitative methods, such as the previously discussed verbal protocol studies, collect data through qualitative observations of behavior by researchers or subjects themselves. Quantitative methods, on the other hand, attempt to collect quantitative data that reflects behavior without introducing potentially subjective human judgment into the data collection process.

Quantitative Evidence

Larger-scale studies, for instance by [Alqadi and Maletic, 2017] and [Ahmadzadeh et al., 2005], analyzed numerical features extracted from students' code submissions. These studies revealed several interesting findings:

- There is not always a strong correlation between general programming ability and debugging skill.
- Some types of bugs are inherently more difficult to debug.
- More experienced programmers tend to be more efficient at debugging.

[Alqadi and Maletic, 2017] created two experiments with 142 subjects in total, but the only data recorded from each subject was the amount of time it took them to debug each bug. Time-elapsed, as a measure of debugging process, does not reflect most details of the debugging process other than overall struggle.

[Ahmadzadeh et al., 2005] collected slightly more data, including information on every compilation error faced by subjects working on debugging tasks, but the primary insight of the experiment was the fact that strong programmers may be weak at debugging. This is a useful insight, but does not reveal any specific details of the debugging process.

In [Alaboudi and Latoza, 2021], the authors leveraged process data in order to identify a behavioral pattern of editing and running code while programming and debugging. This process data was not automatically collected, but instead labeled by humans who observed eleven programmers live-streaming development activities.

The authors identified a cycle between editing code and running it, similar to the main debugging loop of locate-error, repair-error, and run-program identified in previous theoretical models. While some form of quantitative data was used in these experiments, limitations of the methodologies employed in all three experiments prevented analysis from revealing details about the debugging process across a large population.

1.1.3 Research focused on novice debugging

Particular attention has been paid to the debugging processes of novice programmers. Novices are typically worse at debugging than more experienced programmers [Gugerty and Olson, 1986]. As

debugging is an important skill for programmers to develop, understanding the differences between how novices and experts debug is crucial for educators looking to improve the debugging abilities of their students.

Because modern learning management systems produce a lot of data that can be analyzed to understand debugging, there has been some research analyzing process data to better understand debugging amongst novice programmers. This process data can take a variety of forms, but a shared characteristic is that the data points are collected throughout the span of time that students are working on a programming assignment. Process data also describes the process through which a student progresses towards the final solution of their assignment.

[Becker, 2016] proposed a new metric, repeated error density (RED), for measuring student struggles with debugging by looking at repeated compiler errors.

In [Liu and Paquette, 2024] and [Liu and Paquette, 2023] researchers exploited existing submission data to investigate debugging patterns of students. They found that certain features extractable from the submission log data correlates with strategies used by the students for debugging. These strategies were correlated with the efficiency of the students' debugging process. Students were found to be using efficient debugging strategies more as they progressed later in the semester.

One of the largest existing datasets of process data comes from [Edwards et al., 2023]. While not specifically focused on the task of debugging, quite a bit of research has used this data to analyze how novice programmers in a college setting write code. To our best knowledge, none of this work has been specifically focused on the process of debugging. The review article presented by [Edwards et al., 2023] summarizes several threads of current research on the uses of process data in programming education.

1.1.4 Differences Between Experts and Novices

Research has consistently shown that experts and novices differ in their approach to debugging:

- [Vessey, 1984] found that experts are better at chunking program information and tend to follow a breadth-first search strategy.
- Novices, on the other hand, often exhibit more erratic behavior and less systematic code-tracing.

Eye-tracking studies by [Lin et al., 2016] further highlight that high-performing debuggers trace code in a top-down manner, whereas low-performing students struggle to perceive the overall structure of the program.

1.2 Cognitive model used in this research

Based on the literature, we used the following cognitive model of debugging as a foundation for our analysis:

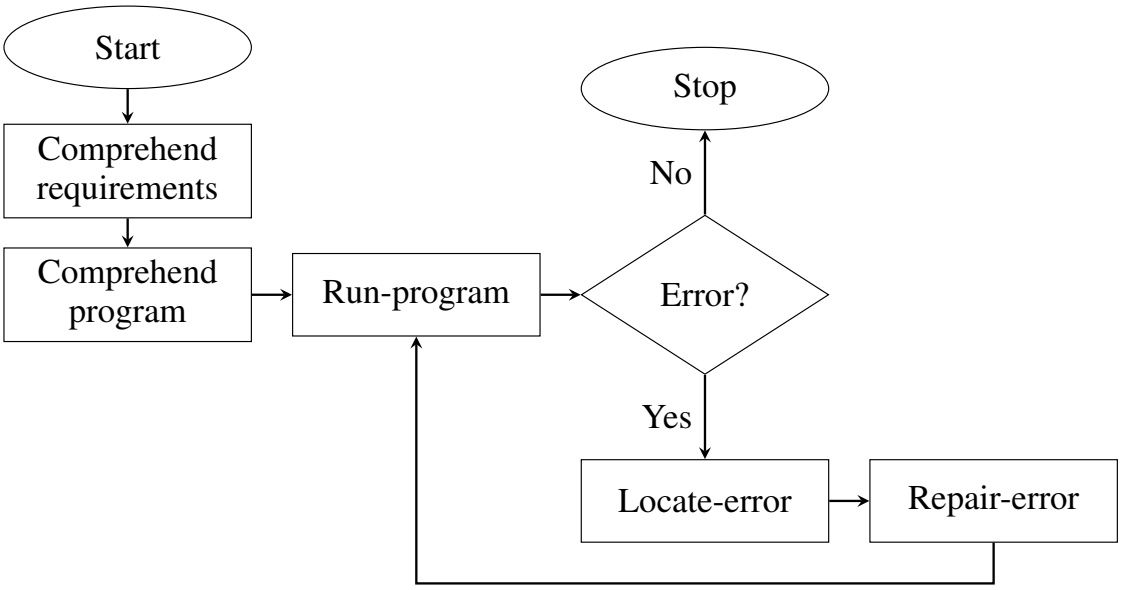

Figure 1.3: **Cognitive model of the debugging process**

This cognitive model closely adheres to the version used in [Kessler and Anderson, 1986], [Vessey, 1984], [Katz and Anderson, 1987], and [Carver and Klahr, 1986].

1.2.1 Cognitive phases of debugging

In order to demonstrate the cognitive phases of debugging, imagine a student working on a debugging exercise. The student is given a set of requirements for a program, as well as buggy code with one or more errors that causes it to not fulfill the requirements. In that context, the cognitive phases of debugging would look like this:

1. **Comprehend problem:** The student begins the debugging process by reading the prompt for the problem. The prompt explains the desired behavior of the program.
2. **Comprehend program:** The student then reads the existing buggy code for the program. The student may take note of control structures, variables, and the flow of execution. This step may include code tracing.
3. **Test program:** The student runs the current version of the code to see if it successfully passes the exercise.
4. **Detect error:** Code execution may reveal an error in the behavior of the program, manifesting as either a runtime error or a failed test case.
5. **Locate error:** The student reads the evidence of the error and attempts to locate where the faulty line of code is within the program.
6. **Repair error:** The student makes a change to the program in an attempt to fix the error they found in the previous step.

1.3 Gap analysis

The literature reviewed in this thesis demonstrates that while early models provided a valuable framework for understanding debugging, more details on the debugging process may be revealed by closer analysis. While a wide variety of methodologies exist, from verbal protocols to large-scale quantitative analyses, there are gaps in existing research. To our knowledge, no experiments have been conducted at a large scale (> 200 subjects) with a specific focus on debugging.

Moreover, the differences observed between experts and novices suggest that effective debugging might be more about strategic thinking and the ability to see the "big picture" than about technical skill alone. Integrating these insights could lead to improved educational methods and debugging tools.

Existing research that uses process data to analyze debugging tends to use coarse measurements - data points are only recorded when students attempt to submit their assignments. Additionally, we were not able to find high-quality process data measured from an isolated debugging exercise: existing research analyzes debugging by slicing out debugging time from the entire timeline of a student working on an assignment.

This poses a few problems - when students are debugging code that they just wrote, each one is debugging different bugs in different code. Thus, it becomes difficult to ascertain what features of the debugging process are caused by a student's lack of debugging skill and what is caused by the inherent difficulty of the bug. Additionally, [Katz and Anderson, 1987] found that programmers debug their own code differently from how they debug another programmer's code.

Thus, a robust analysis of the debugging process over a wide scale must leverage quantitative data from subjects who are working on the same bugs. We believe that analyzing the debugging process using process data is the best method to meet these constraints.

Chapter 2

Problem and approach

The goal of this research is to establish a foundation for using process data to analyze the behavior of programmers while debugging. In our context, process data takes the form of an event log - throughout the process of debugging a program, programmers perform several different types of actions that are logged by our system.

Our core research questions are:

1. Can we determine which cognitive phases programmers spend time in using automated analysis of process data?
   Answer: Yes
2. Does process data reflect debugging skill for students in an introductory CS course?
   Answer: Yes
3. Can we use process data to better understand how programmers debug?
   Answer: Yes

The first step of our approach was to collect a large amount of process data from programmers in the process of debugging. We collected this data from introductory programming students who worked on debugging exercises in Python.

In order to answer the first question, we created a rule-based method of assigning cognitive phases to the timeline of a programmer's debugging process. We also used a panel of experts to validate our automated analysis.

For the second question, we identified a correlation between a measure of debugging struggle and the scores received by students on programming exams.

For the third question, we identified a number of patterns in the process data which raise questions about the debugging process.

Chapter 3

Methods

3.1 Apparatus and instrumentation

The data for this research was collected from an introductory computer science course at Carnegie Mellon University, called "15-112: Fundamentals of Programming and Computer Science". The students in the course are mostly undergraduates (a small minority are graduate students), and the course is their first experience with programming at the college level. The course is taught in Python.

Students in the course complete weekly homework assignments consisting of several Python programming exercises. These homework exercises are conducted in an online, browser-based learning platform that provides an IDE-like code editor for students to write and run their code within the platform. At the time of data collection, students were expected to be familiar with basic concepts in Python programming, including functions, variables, loops, 1-dimensional lists, and strings.

The course included one short lecture on debugging in the first week. That lecture focused on proper usage of print statements while debugging Python code.

3.1.1 Anonymization

All of the data collected for this research was anonymous - students were identified with a random hash that cannot be tied back to their real identity. This research was determined to be exempt from Institutional Review Board (IRB) approval by our IRB because no personally identifying information was collected.

3.1.2 CMU CS Academy

15-112: Fundamentals of Programming and Computer Science (the course from which our data was collected) uses a platform called CMU CS Academy for homework. Every week, students are assigned several programming exercises. Normal programming exercises ask students to write a program given a set of requirements. Students receive a blank function and a set of test cases to start, and must write code such that the function passes the given test cases. An example view of what a student might see on their screen while in CMU CS Academy can be seen in Figure 3.1.

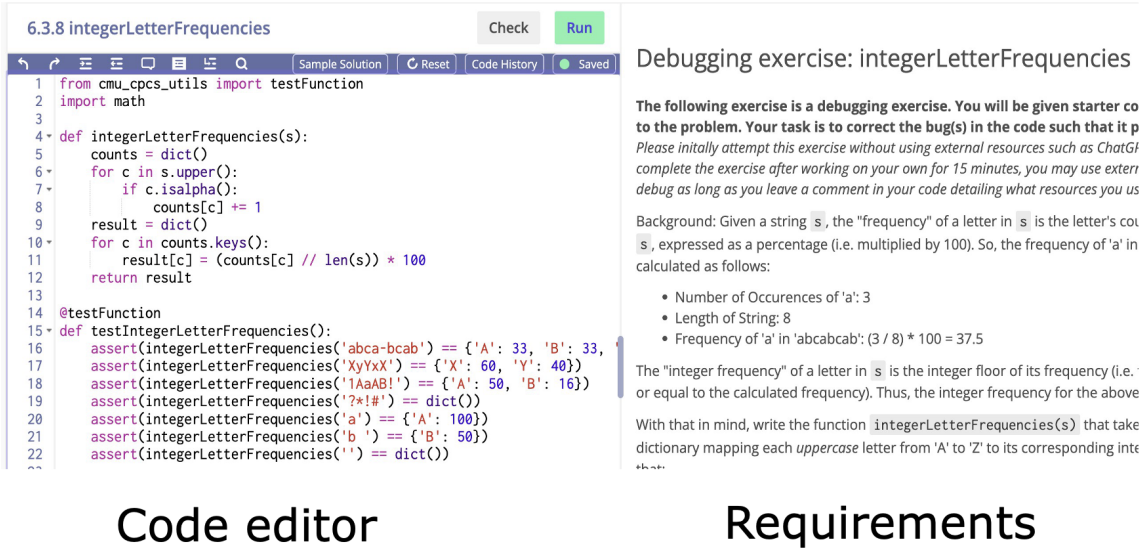

Figure 3.1: **Example view of a CMU CS Academy exercise.**
Students see a code editor on the left, and program requirements on the right. Students are given test cases on which their programs are graded, visible as the testIntegerLetterFrequencies function in the code window.

3.1.3 Process data

CMU CS Academy collects process data from students working on their programming exercises. The process data collected from the platform is output in three separate time series.

1. **Keystroke events:** Every editing event that the student makes in their code editor while they work on their exercise. Editing events are either additions or deletions of code. This data is tracked down to the granularity of keystrokes, so every time a student presses a button on their keyboard to edit their code, it is logged as an event. For each keystroke, the location of the edit, whether or not it was an addition or deletion, the characters added/deleted in the edit, and a timestamp of when the edit occurred are logged. An example snippet of keystroke data is shown in Figure 3.2. Note that events are logged down to the individual character, so when the student types the token "digit," it gets logged as five separate addition events.
2. **Execution events:** In the process of completing an exercise, students can execute their code to see the output. Execution events are recorded when students execute their code - these are shown in the data as `run_code_exercise` events. Events are also recorded when the Python environment used to execute the code returns some output to the student. This output can either be a runtime error, a failed test case, or a pass, which indicates that the student has successfully passed all test cases. These events are logged as `runtime_error`, `sample_test_incorrect`, and `autograde_correct` respectively. Execution events are tagged with a timestamp similar to that of keystrokes.
3. **Access events:** Students were free to work on these exercises on their own time, because

they were assigned as homework. Students could thus open and close the browser window on which they edited their code at any point in time in the week they were given to work on the exercises. Students could also navigate away from the browser tab on which they were editing code without closing their coding tab. These events are logged in our process data - precise timestamps were recorded for every time a student opened up the exercise (`start_session`), closed the exercise (`end_session`), moved their computer's focus away from the exercise's browser tab (`blur`), and returned their computer's focus to the exercise's browser tab (`focus`). This data allows us to determine exactly when students had the exercise up on their screen, in focus.

Start	End	Action	Characters added/deleted	Timestamp
{'row': 3, 'column': 4}	{'row': 3, 'column': 13}	Remove	['return 42']	1734416413175
{'row': 3, 'column': 4}	{'row': 3, 'column': 5}	Insert	['d']	1734416413585
{'row': 3, 'column': 5}	{'row': 3, 'column': 6}	Insert	['i']	1734416413661
{'row': 3, 'column': 6}	{'row': 3, 'column': 7}	Insert	['g']	1734416413796
{'row': 3, 'column': 7}	{'row': 3, 'column': 8}	Insert	['i']	1734416413864
{'row': 3, 'column': 8}	{'row': 3, 'column': 9}	Insert	['t']	1734416413956

Figure 3.2: **Example keystroke data snippet logged by the learning platform.** Note that this information is enough to reconstruct the intermediate state of the program during editing after every keystroke. *Start* and *End* indicate the position of the cursor within the file before and after the edit occurs. *Action* indicates whether the edit was an insertion or deletion. *Characters added/deleted* indicates the characters added or deleted in the editing event. *Timestamp* indicates the time measured in milliseconds since the Linux epoch (Jan 1, 1970).

3.1.4 Debugging exercises

A debugging exercise, in the context of this research, is a coding exercise where the student is given a coding problem that is incorrectly solved because it fails to pass the requirements. The defective code is generated by taking a correct solution to the exercise, and introducing bugs that break the functionality of the correct solution.

These debugging exercises differ from regular programming exercises in that the student is not asked to write an entire program from scratch. The "starter code" that students are given to start the exercise only differs from a correct solution by a few tokens. A student could feasibly complete the exercise by modifying a very small number (≈ 5) of tokens in the code.

Three debugging exercises were used for this experiment: bowlingScore, moveToBack, and integerLetterFrequencies. bowlingScore and moveToBack were assigned to students in week 5 of the semester, while integerLetterFrequencies was assigned in week 9.

To create the debugging exercises, we started with existing programming exercises. In previous semesters, students have been asked to complete these exercises by writing a program from scratch. We introduced two bugs into each program.

A summary of each exercise is shown in Table 3.1. The complete details of each exercise, including requirements and code, can be found in the appendix.

Exercise	Started	Completed	Unit	Description	Week
bowlingScore	212	153	1d Lists	Sum bowling scores	5
moveToBack	309	268	1d Lists	1d list manipulation	5
integerLetterFrequencies	315	310	Dictionaries	Count frequency of letters in a string	9

Table 3.1: **Summary of debugging exercises**

Choosing an exercise

For the week 5 exercises, bowlingScore and moveToBack, students were able to choose to complete one exercise or the other. Completion of only one exercise was required to receive homework credit. Because of this, not all students completed both of these exercises.

On the other hand, all students were required to complete integerLetterFrequencies to receive homework credit, resulting in a higher completion rate.

3.2 Subjects

319 students attempted to solve at least one of the three debugging exercises used for this experiment. 212 attempted to solve the bowlingScore exercise, 309 attempted to solve the moveToBack exercise, and 315 attempted to solve the integerLetterFrequencies exercise.

3.2.1 Instructions to subjects

Students were told about the debugging exercises through the same channel through which they receive their homework assignments every week. Students were given the following instructions to complete the exercises for this experiment:

> *Instructions: Please try to fix these solutions on your own at first. If you use external resources, please note this in a comment in the code. This code was not necessarily written with 112 style in mind, and these two problems will not be graded on style. We hope you will not spend more than 15 minutes total on this task. After this time, you may stop if you wish, and you will still get the homework points if you haven't fixed the code. Note: This is an experiment to provide you with debugging practice, and to provide the CS Academy team with valuable research data.*

The above snippet is the first thing that students read on the assignment page. After that page was shown, students navigated to the exercise page, which contained the prompt for the exercise as well as the code editor.

The following instructions were given to subjects at the top of each exercise prompt:

> *The following exercise is a debugging exercise. You will be given starter code with a buggy solution to the problem. Your task is to correct the bug(s) in the code such that*

it passes the test cases. Please initially attempt this exercise without using external resources such as ChatGPT or your peers. If you cannot complete the exercise after working on your own for 15 minutes, you may use external resources to help you debug as long as you leave a comment in your code detailing what resources you used.

After these descriptions of the debugging task, students were given the requirements for the program as they were in the original (non-debugging) exercise.

3.3 Data

The data collected for this research comes from the learning platform on which students complete their exercises. We collected process data, consisting of sequences of events observed through the students' browser window while they were working on the exercises.

3.3.1 Intermediate code states

An important property of our process data is that it reveals the exact contents of a subject's code editor at every point in time. Each event in the keystroke event log represents one change made to the code in the code editor.

Considering the keystroke log snippet in Figure 3.2, if we know that the initial state (starter code) of the snippet was

```
return 42
```

we know that after the first event (a deletion of the string "return 42" the code editor was empty. Subsequently, after the next five events, we know that the code editor showed

```
digit
```

because of the addition events.

3.3.2 Token-level changes

The collected keystroke event log records changes at the character level. Thus, a student typing the word "digit" shows up as five separate keystroke events. For ease of analysis, we would like the addition of one token to correspond to one event. This has the advantage of allowing us to categorize edits based on the token being edited.

The implementation of token-level edits is shown in Algorithm 1. The algorithm makes use of the Python tokenizer, which can output a stream of tokens given text. Each character-level edit event may add, delete, or change some number of tokens in the code. To get token-level edits, we merge adjacent edits to the same token, until no two consecutive edits are on the same token.

Algorithm 1 Character-to-Token Edit Transformation

Require: Initial source code S_0, character edits $E = \{e_1, e_2, \ldots, e_n\}$
Ensure: Token-level edit sequence T

$states \leftarrow$ reconstruct code states from S_0 and E
$validStates \leftarrow$ filter $states$ for syntactically valid code
$tokenStates \leftarrow$ tokenize each state in $validStates$
$currentTokens \leftarrow$ tokenize(S_0)
$rawDiffs \leftarrow []$
for each $nextTokens$ in $tokenStates$ **do**
  $diff \leftarrow$ enhanced_diff$(currentTokens, nextTokens)$
  append $(timestamp, diff)$ to $rawDiffs$
  $currentTokens \leftarrow nextTokens$
end for
$T \leftarrow$ merge_adjacent_changes$(rawDiffs)$
return T

Merge Subroutine:

$merged \leftarrow []$, $current \leftarrow null$
for each $diff$ in $rawDiffs$ **do**
  if $diff$ has exactly one token change **then**
    if $current = null$ or $current.index \neq diff.index$ **then**
      append $current$ to $merged$ (if not null)
      $current \leftarrow diff$
    else
      $current.new\_token \leftarrow diff.new\_token$ {extend change}
    end if
  else
    append $current$ to $merged$ (if not null), $current \leftarrow null$
    append $diff$ to $merged$
  end if
end for
append $current$ to $merged$ (if not null)
return $merged$

Helper Functions:

- `ReconstructCodeStates`(S_0, E): Applies character-level edits E to initial source S_0 to reconstruct intermediate code states with timestamps
- `ParseAST`$(code)$: Parses source code into Abstract Syntax Tree, returns null if syntax error
- `TokenizeSource`$(code)$: Converts source code into sequence of tokens
- `DiffTokens`$(tokens_1, tokens_2)$: Compares two token sequences and returns tuple of (added_count, deleted_count, changed_count)

3.3.3 Functional and print changes

A student starts a programming exercise with the starter code provided to them, and hopefully finishes an exercise with a correct solution. In between these states, the student makes a number of changes at different points in time in order to go from starter code to correct solution.

These changes can be represented as edits to the sequence of tokens of the program. We classify changes to the code into two buckets: print changes and functional changes. Print changes (in Python) are changes to the code that only change parts of the program in print statements. This includes adding print statements, removing print statements, and modifying what is printed. For example, a change from:

```
print(scores[i])
```

to:

```
print(scores[j])
```

is classified as a print change that changes one token (in this case changing the element of `scores` that is printed).

Functional changes are all other changes. These are changes that, in theory, should modify the behavior of the program. For example, the following change from:

```
def some_function(a):
    return a + 1
```

to:

```
def some_function(a):
    return a + 2
```

would be classified as a functional change. Note that functional changes may not actually affect the input and output of a function - for example, the student may edit code that turns out to be dead. Going from:

```
a = 1 / 0
return b
```

to:

```
a = 1 / 0
return b + 1
```

would still be classified as a functional change, even though it does not materially impact the behavior of the program at execution: execution will never reach the return statement, as there is a divide-by-zero error on the first line. However, we still see this change (and subsequent execution) as an attempt by the student to test new functionality of the code, even if the change is ultimately unsuccessful in repairing the error.

The two types of changes are differentiated by taking the code before and after the change, removing all print statements from the both versions of the code, and determining if the two versions are identical after removing all prints.

3.3.4 Functional and non-functional execution

Based on this notion of functional and print changes, we can categorize each execution of the code by the student as either functional or non-functional. Functional executions occur when there has

been a functional change in between the state of the code at the previous execution (or in the case of the first execution, the starter code) and the state at the current execution. A non-functional execution occurs when the student runs their code with no functional changes since the last time they ran their code.

For example, if a student ran the following code:

```
print(scores[i])
scores[i] = 5
return scores
```

and at the next execution, the state of the code was:

```
print(scores[i])
scores[i] = 6
return scores
```

We would categorize the second execution as functional. If the next (third) execution was run with the following code

```
print(scores[j])
scores[i] = 6
return scores
```

We would classify the third execution as non-functional, as there are no functional changes between the code at the second and third executions.

3.3.5 Extracted event log

In order to better analyze the process data, the following features were extracted from the collected data.

1. **Intermediate code states:** When a student first opens up an exercise, the code in their editor is the starter code. Every keystroke event changes the state of the code in their editor. The intermediate state of the code in the student's editor is extracted by continually applying the effects of each keystroke event in sequence, starting with the starter code, in order to reconstruct exactly what code the student had in their editor at every point in time.
2. **Print changes and functional changes:** Using the information in the intermediate code states, each keystroke event was tagged as being either a print change or a functional change.
3. **Functional and non-functional executions** Using the intermediate code state at the time of each code execution, each execution was tagged with whether or not it was a functional or non-functional execution.

Thus, the ingested data was processed into a combined event log of 11 different kinds of events:

1. `start_session`
2. `end_session`
3. `focus`
4. `blur`
5. `functional_change`
6. `print_change`

7. `functional_execution`
8. `nonfunctional_execution`
9. `runtime_error`
10. `sample_test_incorrect`
11. `autograde_correct`

A visualization of these events for two exercise attempts can be seen in Figure 5.1 and Figure 5.2. We call this event log the **extracted event log**.

Chapter 4

Extracting Cognitive Phases from Process Data

Cursory examination of our extracted event logs, such as the visualization in Figure 5.2, indicates that details on the student's process of debugging are indeed hidden in the process data. In order to analyze this process at a higher level of abstraction than edits and executions, one of our goals was to associate a student's extracted event log with a sequence of cognitive phases. These cognitive phases (as explained in Section 1.2.1) represent the distinct tasks programmers must complete in the process of debugging.

4.1 Approach

More formally, our goal was to partition the time a student was working on a debugging exercise into separate cognitive phases, based on the information in the extracted event log. This equates to identifying *transition points* - points in time at which the student moved from one cognitive phase to the next.

The cognitive model of debugging first shown in Figure 1.3 is reproduced below for convenience.

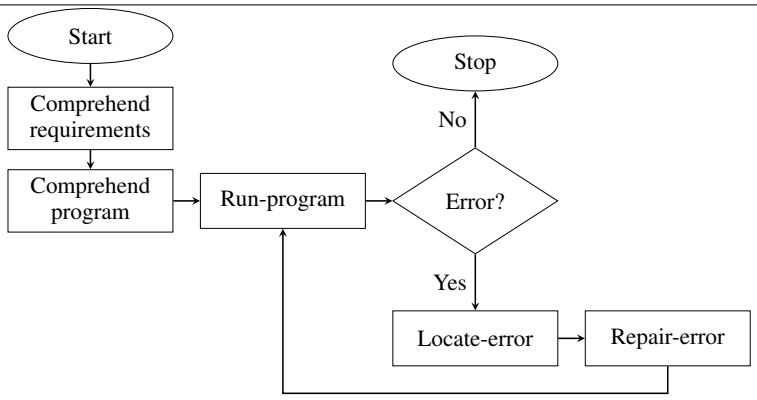

Figure 4.1: **Cognitive model of the debugging process**

Our goal was to associate each transition (represented as an arrow in Figure 4.1) with a marker in the extracted event log. For our purposes, a marker is an event or series of events in the extracted event log.

We excluded the "Comprehend requirements" phase from our analysis, as it occurs before process data is first recorded - students could read the requirements for a debugging exercise before opening up the code editor. Additionally, run-program phase occurs relatively instantaneously, so we did not need to identify a marker for exiting run-program phase: this happens automatically once the program finishes execution shortly after it starts, so we assume locate-error phase begins immediately.

This left us with three transitions for which we needed to identify markers:

1. Comprehend program → run-program
2. Locate-error → repair-error
3. Repair-error → run-program

We noted that run-program phase and repair-error phase are *concrete*: they can be directly observed in the extracted event log. Students running programs to test out new functionality corresponds exactly with our notion of functional executions, and students attempting to repair errors must be making functional edits. Students running the program for the first time may be performing a non-functional execution by running the starter code unmodified.

Thus, we created the following set of markers:

1. Comprehend program → run-program: **functional execution** or **non-functional execution**
2. Locate-error → repair-error: **functional edit**
3. Repair-error → run-program: **functional execution**

4.2 Panel of Experts

In order to validate our approach for identifying cognitive debugging phases from extracted event logs, we conducted an panel-of-experts study. This section describes the methodology and results of this evaluation.

4.3 Participants

We recruited five experts in computer science education to participate in our evaluation:

- Four experienced undergraduate teaching assistants from the introductory computer science course that was the source of our data
- One professor who has taught 15-112 (the course used for this research) in the past, and currently teaches an introductory programming course aimed at students who are not majoring in computer science.

All experts had extensive experience helping novice programmers debug code and were familiar with the common patterns and challenges faced by introductory programming students. Each expert had at least two semesters of experience working directly with students on debugging tasks.

4.4 Materials

Experts were provided with:

- Background information on the event log format and event types (as described in Section 3.1.3)
- The cognitive debugging phase model (as described in Section 1.2)
- Two extracted event logs from actual student debugging sessions:
  - Event Log 1: A 1,533-second session with 39 events, representing a student who struggled with multiple attempts before successfully fixing all bugs
  - Event Log 2: A 657-second session with 11 events, representing a student who efficiently identified and fixed both bugs with minimal attempts
- A structured form for labeling cognitive phases between each event and rating their confidence in these labels

4.5 Procedure

Experts were instructed to:

1. Review the background materials on debugging phases and event types
2. For each extracted event log, label the cognitive phase(s) they believed the student was engaged in between consecutive events
3. Provide qualitative feedback on their labeling process and any patterns they observed, as well as their confidence in their labels. To gauge their confidence, experts were asked the following question: "*How confident are you that your labels correspond to the actual cognitive phases of the student who did this debugging assignment, on a scale of one to five? One is not confident at all, and five is very confident.*"

Experts were explicitly instructed to consider the Test Program (TP) phase as a point event coinciding with run events, while the other phases (Comprehend Program, Locate Error, and Repair Error) should be labeled as intervals. They were asked to provide a complete labeling that covered all time the student spent focused on the program.

4.6 Analysis

To quantify the agreement between our automated phase labeling and expert labeling, we computed the Levenshtein distance (L.D.) between the sequence of phases. The L.D. measures the number of events in total (across both event logs) for which the expert disagreed with the automated cognitive phase labeling.

4.7 Results

4.7.1 Agreement with Automated Labeling

As shown in Table 4.1, the expert labels closely matched our automated labeling approach, with an average Levenshtein distance of 3.0 (σ = 1.4).

Expert	L.D.	Confidence Score
Expert 1 (TA)	2	4
Expert 2 (TA)	2	4
Expert 3 (TA)	2	4
Expert 4 (TA)	0	4
Expert 5 (Professor)	5	3

Table 4.1: **Agreement between expert and automated cognitive phase labeling.** L.D. (Levenshtein distance) measures the edit distance between the sequence of phases. Confidence scores indicate experts' self-rated confidence in their labels (1-5 scale).

4.7.2 Inter-rater Agreement

The average pairwise Levenshtein distance between expert labels was 1.4, indicating strong agreement among experts about the cognitive phases students were engaged in during the debugging process. Disagreements between the expert labelings and the automated labeling were limited to differences in whether the locate-error phase started at the beginning or end of program execution: an unimportant distinction for our purposes, as program execution generally lasted for a small fraction of a second.

4.7.3 Qualitative Feedback

Experts consistently noted a relatively simple pattern in their labeling process:

- Students start in comprehension phase, then move to locate-error phase.
- The first functional change switches them to repair-error phase.
- The functional execution ends repair-error phase and constitutes a run-program phase.
- The student immediately goes back into locate-error after an error.

This labeling strategy corresponds closely with our own method of assigning cognitive markers to event logs.

4.8 Cognitive phase log

Our automated labeling returns a list of intervals corresponding to cognitive phases of debugging. This timeline shows when each cognitive phase begins and ends during the debugging episode. A

visualization of an example timeline can be seen in Figure 4.2.

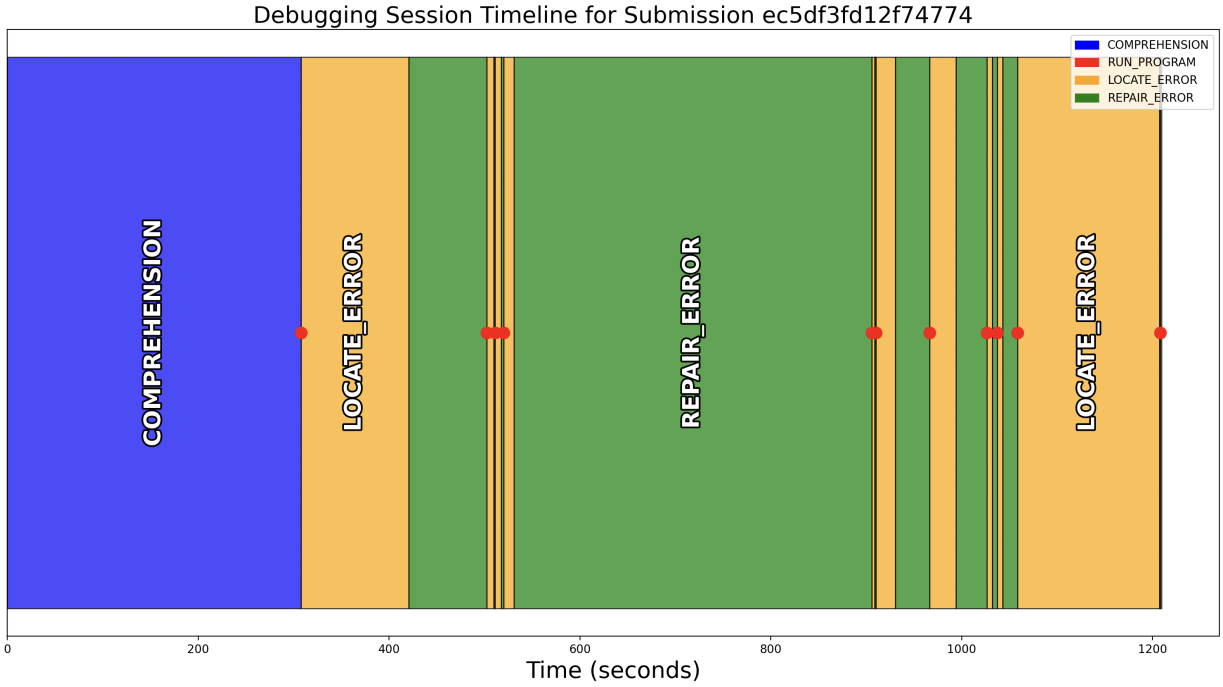

Figure 4.2: **Extracted cognitive phase timeline for a student submission to the bowlingScore exercise.**
Each colored box represents one cognitive phase, and the whole timeline represents one student's work on the bowlingScore exercise. Note that the length of cognitive phases varies significantly, from a few seconds to several minutes.

4.9 Discussion

The high level of agreement between experts and our automated approach validates our method for extracting cognitive phases from debugging event logs. High levels of confidence indicate that these labels are likely to correspond to underlying cognitive phases to at least some degree.

The primary source of disagreement concerned transitions between locate-error and repair-error phases, particularly when students made small edits in close succession. The lack of detailed information as to edit contents provided to the panel led some of the panelists to believe that locate-error and repair-error phases may in reality overlap, with the student performing both tasks simultaneously.

These areas of disagreement align with the theoretical complexity of precisely delineating cognitive phases - our cognitive model makes the simplifying assumption that the student is in exactly

one phase at every point in time.

Previous research has indicated that programmers tend to debug in distinct cognitive phases in practice [Katz and Anderson, 1987]. The cognitive phases of debugging are also neurally distinct as detected by fNIRS [Hu et al., 2024].

4.10 Conclusion

Our expert evaluation provides strong validation for our automated method of assigning cognitive phases to programming process data. The high agreement between the experts and our automated approach suggests that behavioral markers in event logs can effectively detect student cognitive states during debugging activities.

Chapter 5

Analysis of Patterns in Process Data

Our process data was collected at a very large scale, with more subjects and more data collected per subject than any previous debugging study we were able to find. One of our objectives with this data was to explore patterns that appeared in the debugging process across our subject population, through our process data. These patterns revealed new information about how novices approach the task of debugging, and raised questions about the debugging process that could be addressed by future work.

5.1 Case studies

In order to better understand the information available from the extracted event log from a debugging episode, we examine two submissions in detail using the extracted event log.

5.1.1 Student 1

An example timeline of process data events for a student who successfully passed the bowlingScore debugging exercise can be seen in Figure 5.1. This student identified and corrected both of the bugs on line 6, ran their code, and saw the error caused by line 11. The student then wrote code to print the `total` variable, as well as the fix to the bug on line 11, ran their code twice, and passed the tests to complete the exercise.

The student found the bugs quickly and efficiently, only requiring one functional execution to fix both bugs, achieving a correct submission at the end as evidenced by the autograde_correct event. The student clicked on and off the exercise webpage several times, as indicated by the blur and focus events, and only started making changes to the code after about 9 minutes.

While the exact code being edited is not visible through the sequence of events in the timeline, a play-by-play of a student's debugging process can be inferred by the sequence of process data events.

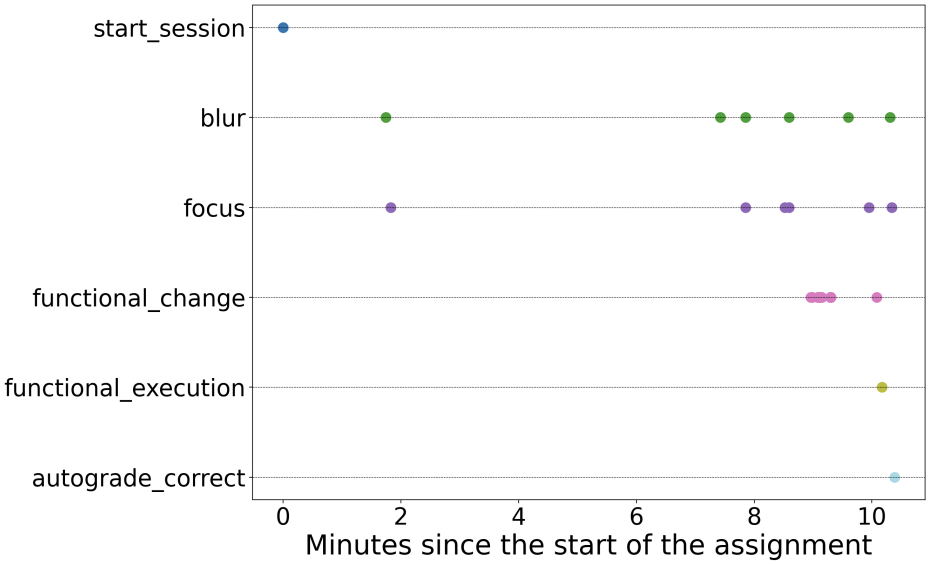

Figure 5.1: **Extracted event log for student 1.**
Each point represents a separate event in the extracted event log, with the event type on the y-axis and the time of the event on the x-axis.

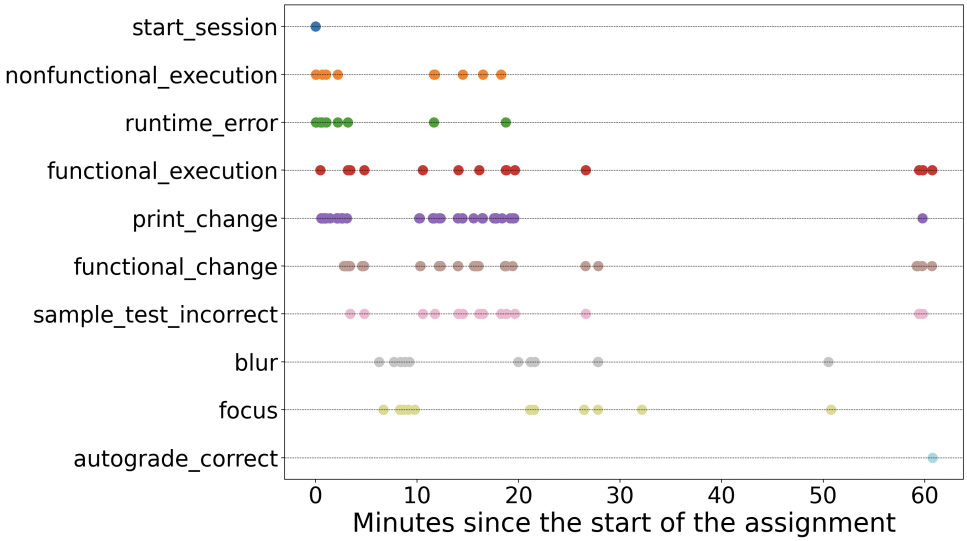

Figure 5.2: **Extracted event log for student 2.**

5.1.2 Student 2

An example extracted event log from a student who struggled a bit more with debugging bowlingScore is shown in Figure 5.2. The student's first changes to the code are print changes, indicating that they added print statements to the code. After a few minutes, the student started to make

functional changes. The error changed from a runtime error to a failed test case, indicating that the student fixed one of the two bugs in the code. After about eight minutes, the student had a few blur and focus events in quick succession - the student went back and forth between the exercise tab in their browser and something else on their computer. Ten minutes after the start of the exercise, the student made more functional and print changes in an attempt to repair the bug, but was unsuccessful in this block of time.

Note the nonfunctional executions - the student ran the code several times with no changes to the code's logic, suggesting that the student was making use of the print statements to observe program behavior. After taking a long break of about forty minutes where the student again switched between the exercise tab and something else on their computer, the student was able to fix the remaining bug relatively quickly about an hour after they started the exercise. Successful completion of the debugging exercise is indicated by the `autograde_correct` event.

The extracted event log shows any given student's path through a debugging exercise with some detail. While it does not show the actual code that students are editing, the events in the log mark important actions taken by the student while debugging.

5.2 Debugging cycles

In order to analyze debugging behavior across the subject population, we formalized the concept of a **debugging cycle** within the context of our process data.

We define a debugging cycle as a round trip around the loop within the cognitive model of debugging, as seen in Figure 5.3.

We used the number of debugging cycles to debug a program as a measure of struggle for a particular debugging episode. A student who needed many debugging cycles to correctly debug a program used more attempts to repair the bug before getting it right. A student who debugged more efficiently would repair the bugs in fewer attempts.

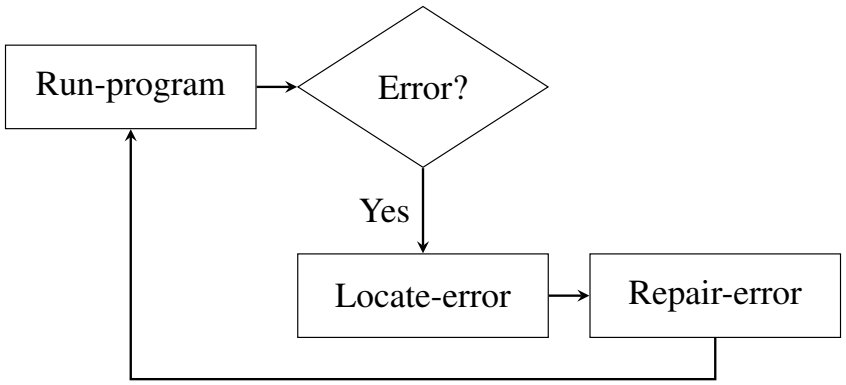

Figure 5.3: **Debugging loop.**
This is a subset of the cognitive model of debugging, consisting of the **locate-error**, **repair-error**, and **run-program** phases. A debugging cycle is one trip around this loop.

For more information on why we did not use time-elapsed as a metric of debugging efficiency, refer to Section 5.7.

5.3 Count of debugging cycles across exercises

The number of debugging cycles used by all students in bowlingScore and integerLetterFrequencies is shown in Figure 5.4.

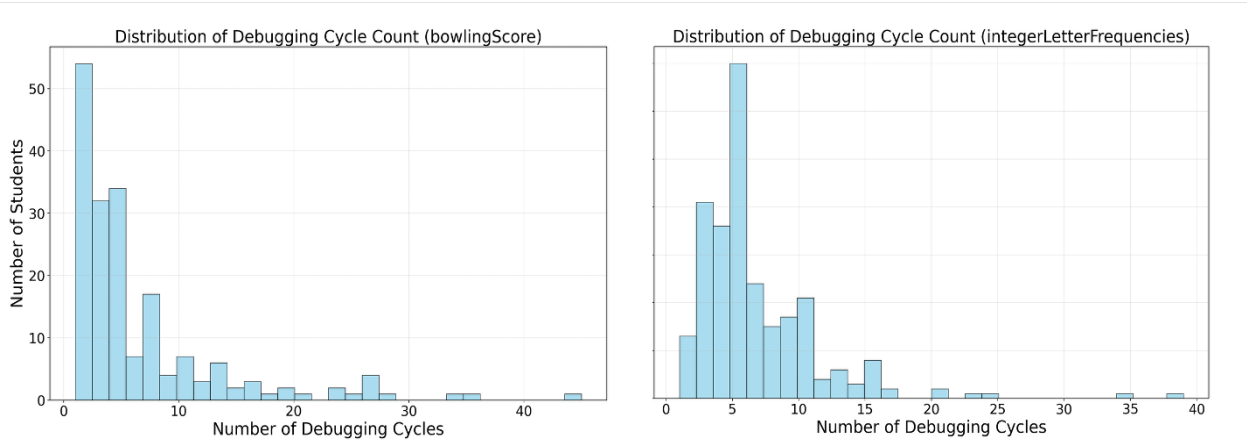

Figure 5.4: **Histogram of number of debugging cycles used by each student in bowlingScore and integerLetterFrequencies exercises**

An interesting detail to note is that the distribution of cycles used does not decrease on average between bowlingScore and integerLetterFrequencies. This is somewhat surprising, as integerLetterFrequencies was given to students four weeks after bowlingScore. In that four weeks, students received no explicit debugging instruction.

One possibility is that students did not significantly improve their debugging skills between the two exercises. This would suggest that programming instruction and practice that is not focused on debugging may not significantly improve debugging skills as an indirect effect.

Another possibility is that integerLetterFrequencies was more difficult for students to complete, even with improved debugging skills. A more robust experiment needs to be conducted to verify the effects of various teaching strategies on debugging skill.

The number of cycles used by students in any given debugging episode varied widely. Most students tended to complete debugging within a few (< 7 cycles), but there was a long tail of students who needed thirty or more cycles to complete debugging. We believe that students who tested functional changes to their code significantly more than the rest of the population could be considered flailing.

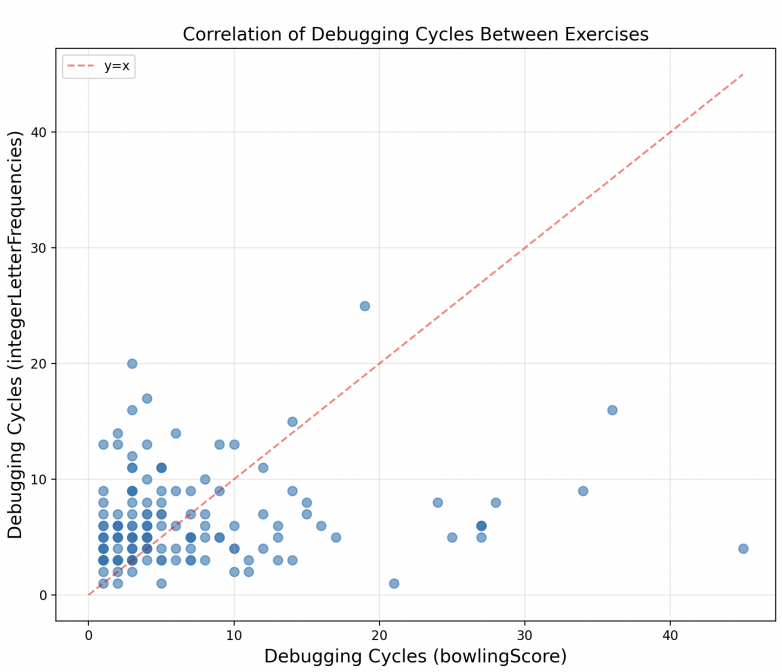

Figure 5.5: **Scatter plot of debugging cycles used in bowlingScore and integerLetterFrequencies for students who did both**

There was no strong correlation between the number of debugging cycles used to complete bowlingScore and integerLetterFrequencies. The population of students who completed both exercises is a subset of the students in the course, since many students completed moveToBack instead of bowlingScore, and about 30 students completed bowlingScore but dropped the course before completing integerLetterFrequencies.

5.4 Time evolution of debugging cycles

Most students went through the repair error phase multiple times while debugging, as seen in Figure 5.4 - each debugging cycle includes one repair-error phase. We examined differences between repair-error phases occurring earlier and later in a student's debugging episode to study how a student's first debugging cycle differs from their fourth, or their tenth.

In general, students complete debugging cycles quicker and with less total repair keystrokes over time. This pattern suggests that programmers gather information about the program in each debugging cycle, which may increase their comprehension of the program. In later cycles, students may have a better familiarity with the program and where to look for errors, which allows them to make quicker and more focused edits.

In any case, this pattern suggests that a disproportionate amount of the time and effort employed in repairing errors occurs early on in the debugging episode, when programmers have spent less time working with the program. Students may not have fully comprehended the program even after exiting the comprehend program phase, skipping to the main debugging cycle before a complete understanding of the program's structure and logic was achieved.

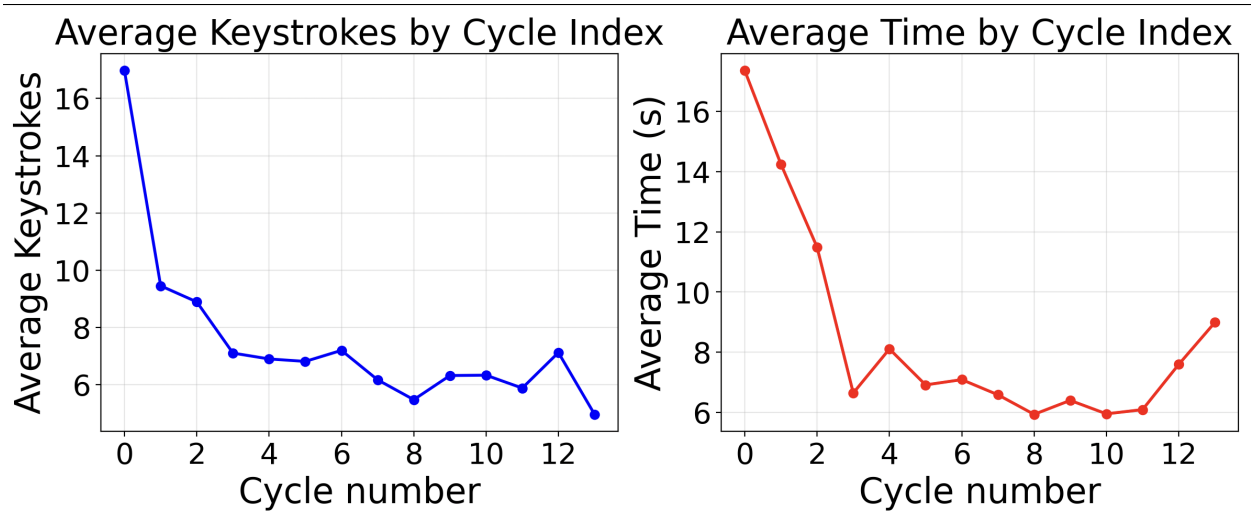

Figure 5.6: **Number of keystrokes and time elapsed in seconds for each repair-error phase, for the integerLetterFrequencies exercise.**
Across student submissions, the first debugging cycle took 20 seconds on average, with around 19 keystrokes inputted on average by each student. Later on in debugging episodes, the amount of time spent and keystrokes inputted in each repair-error phase decreased (on average). Only students who went through four or more debugging cycles and eventually completed the exercise are represented in this graph, for a total of 208 students.

5.5 Correlation with exam scores

Students in the course take exams throughout the semester (three in total). These exams are proctored, and conducted with pen and paper. The exams are designed to test students' knowledge of general programming principles, as well as the Python language.

We found that there was a statistically significant correlation between a student's performance on the integerLetterFrequencies exercise (as measured by number of debugging cycles used) and their average exam grade, as seen in Figure 5.7.

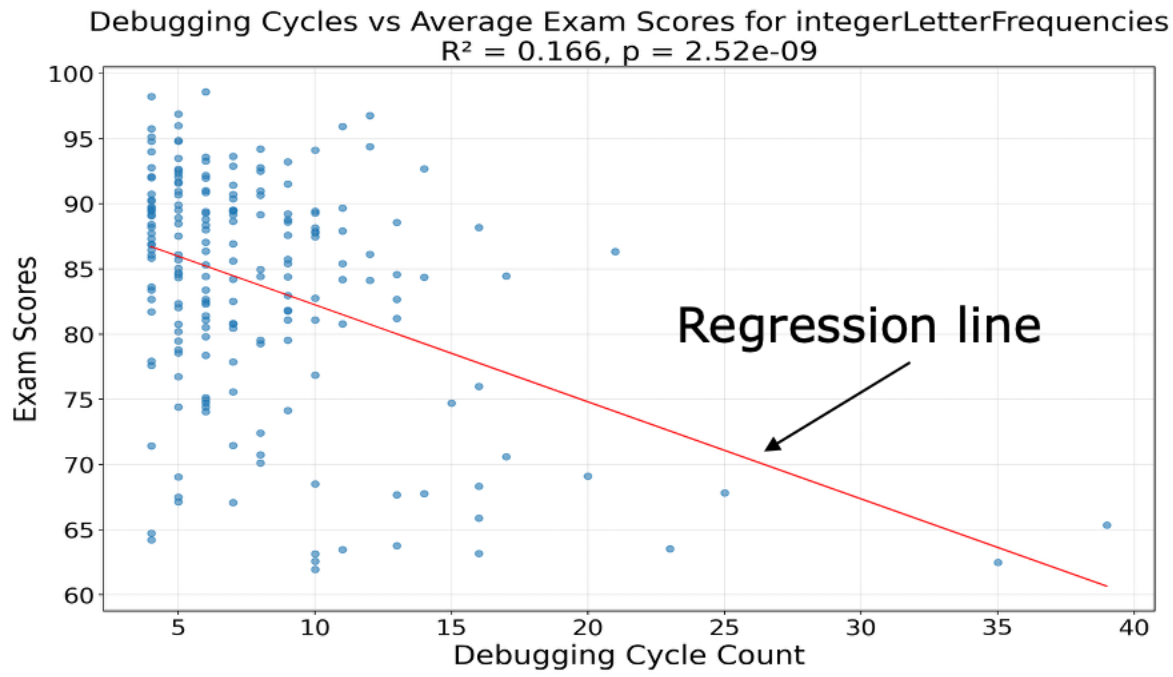

Figure 5.7: **Correlation between debugging cycle count on integerLetterFrequencies and average exam grades.**
Students who used more debugging cycles to debug integerLetterFrequencies tended to get slightly worse grades on exams.

This correlation is weak, which intuitively makes sense - there are many factors that make up a student's understanding of code that are not debugging. [Fitzgerald et al., 2008] found that novices who are good at programming in general may not necessarily be good at debugging. Still, the statistical significance suggests that debugging efficiency has some effect on student exam performance.

5.6 Completion grading

Programming assignments in the course are primarily graded for completion. Students write code to solve the assigned problem, and are graded against test cases. Pass more test cases, and you get more points. We can call this method of grading **completion grading**.

Completion grading can reduce the resolution of feedback provided to both students and instructors. Almost every student (310/315) received full marks on the integerLetterFrequencies debugging exercise, as shown in Figure 5.8. This suggests almost no variation in student performance.

However, a closer look at the graph on the right of Figure 5.8 reveals that the number of debugging cycles used to complete the exercise varied widely. Two students got the same full credit on the exercise: one used two cycles, and the other used thirty-eight.

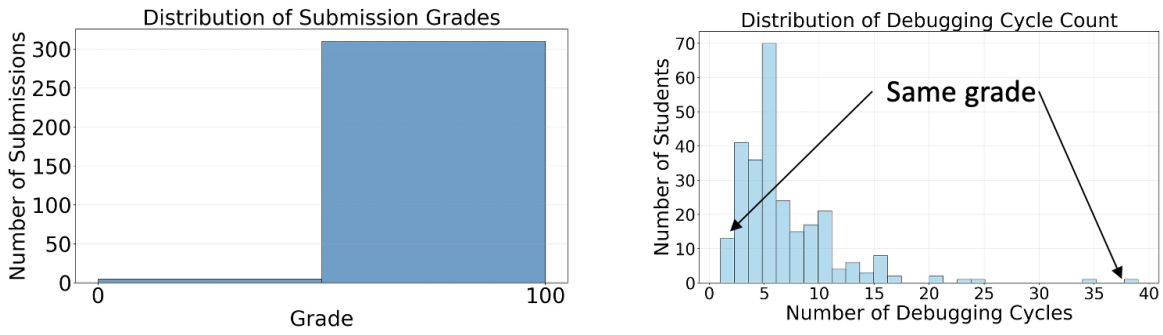

Figure 5.8: Left: **histogram of grades received by students on the integerLetterFrequencies debugging exercise.** Right: **histogram of debugging cycles used to solve the exercise.** Almost every student (310/315) received full credit for the exercise based on completion grading, but the number of cycles used to complete the exercise varied widely.

When standard completion grading fails to measure important details of performance, such as in this case, process data provides a more complete picture of a student's debugging performance than simple completion grading. Focusing solely on the correctness of the results fails to shed light on the process through which students achieved those results. Process data analysis of the sort introduced in this work could provide useful feedback on debugging skill for instructors and students alike.

5.7 The problem with time-elapsed

An obvious metric for student debugging effort is simply time-elapsed: measure how long it takes students to complete a debugging exercise, and that should represent how much effort it took. Unfortunately, this approach has some weaknesses.

Measuring the amount of time a student spends in any particular phase or even for the entire exercise poses problems for statistical analysis. Students worked on these exercises in different places - some may have worked in quiet libraries, while others could have worked in distracting environments. Students worked on exercises at different times of day, as well. Some students may have been looking at their phone while debugging. While these factors probably affect many metrics, metrics involving a measurement of time are particularly susceptible to noise.

Figure 5.9 shows the distribution of total focus time spent in total and in locate-error phase for the integerLetterFrequencies exercise. Two things become apparent:

1. Students at the maximum of the distributions recorded more than 8000 minutes (5.5 days) on the exercise, clearly a measurement error. Many students seem to have left the exercise on their screens for extremely long periods of time.
2. The distributions look very similar in the long tail - this suggests that measurement errors are largely caused by time inaccurately ascribed to locate-error phase.

While the students who spent many hours or even days in locate-error phase present the most extreme problem with time-elapsed measurements, the problem can occur at a smaller scale as

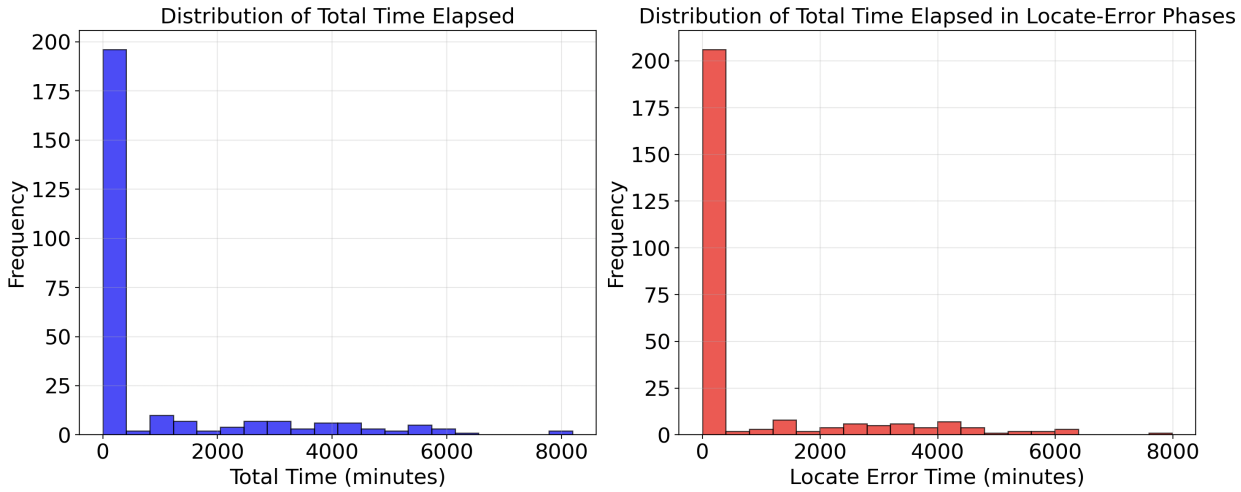

Figure 5.9: **Total time elapsed and total time elapsed in locate-error phase (integerLetterFrequencies)**
Note the long tail of students who ”spent” unrealistically long times in total and in locate-error phase.

---

well. A student who looks at their phone for 30 seconds while they are supposedly in locate-error phase would not necessarily present as anomalous in the process data, but the measurement is still inaccurate.

This is the reason that our metric for student debugging efficiency was not tied to elapsed time, and instead was based on the number of debugging cycles. A student who gets up from their computer may exhibit a spurious increase in time elapsed, but no increase in debugging cycles. This issue may be alleviated by conducting experiments under standardized laboratory conditions, but it is unclear if elapsed time would still reflect debugging skill.

5.7.1 Patterns in time-elapsed across phases

That being said, we do not necessarily have to discard all of our data that measures time. One countermeasure for the noise described in the previous section is to filter out cognitive phases that last longer than five minutes, as these phases are likely to include periods of time where the student is not truly focused on the exercise. Thus, for analyzing metrics related to time-elapsed, we simply exclude all phases longer than five minutes.

Doing so allows us to analyze the amount of time students spend in each cognitive phase while debugging. Figure 5.10 shows the correlation between the total amount of time students spent on integerLetterFrequencies with the total amount of time students spent in locate-error phase. Most of the variation in total time elapsed comes from variation in the time spent in locate-error phase (r=0.845).

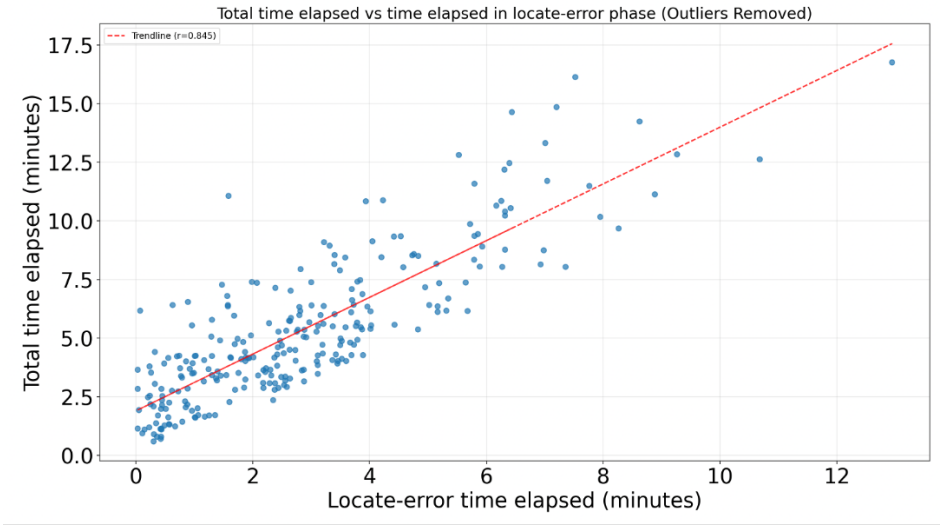

Figure 5.10: **Total time spent vs time spent in locate-error phase (integerLetterFrequencies)**

Figure 5.11 shows the percentage of total time-elapsed spent in each of three cognitive phases. Around half of the time students spent debugging integerLetterFrequencies came from locate-error phase. Repair-error phase and comprehend-program phase took up roughly equal proportions of total time spent debugging.

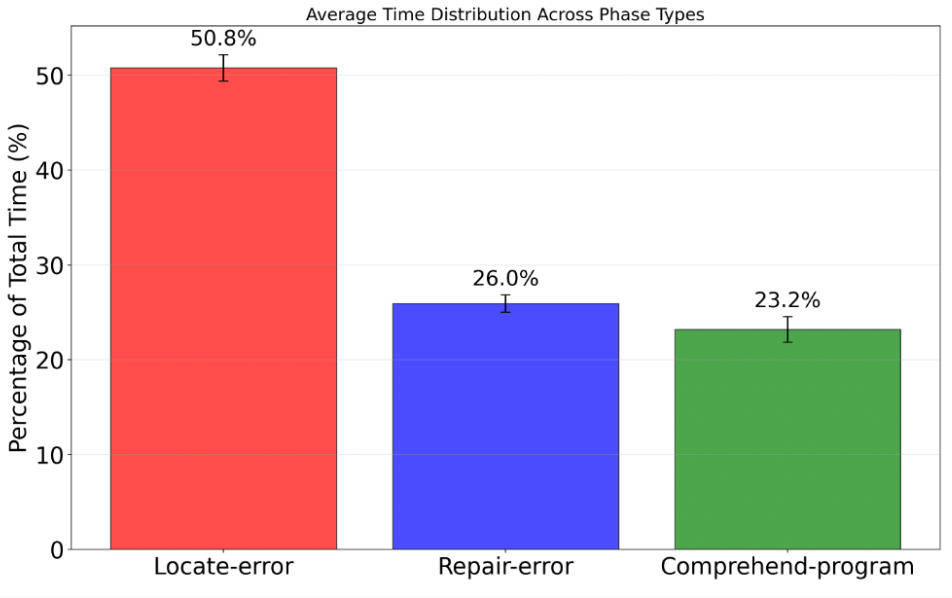

Figure 5.11: **Percentage of time-elapsed in each cognitive phase (integerLetterFrequencies)**

5.7.2 The importance of locate-error phase

To recap:

- Variation in time spent debugging comes (mostly) from locate-error phase.
- Students spent more time in locate-error phase than the other cognitive phases.

The above patterns, shown in Figure 5.10 and Figure 5.11, suggest that locate-error phase may be the most crucial phase of debugging with respect to skill. Students who debugged integerLetterFrequencies quicker were able to do so because they were able to locate the error in less time.

Attempts to improve the efficiency of debugging should focus on improving locate-error phase. Previous research backs up this claim: [Vessey, 1984] found that locating errors was the most difficult and time-consuming part of the debugging process through verbal protocol analysis.

Chapter 6

Identifying efficient and inefficient behaviors

One of the goals of this research was to use process data to identify efficient and inefficient behaviors exhibited by programmers while debugging. Students who are more efficient at debugging may go about the task of debugging differently from students who debug less efficiently. By identifying these efficient and inefficient behaviors, we may be able to inform debugging instruction. Unfortunately, our efforts to use process data to identify the effect of efficient and inefficient debugging behaviors were generally not successful, for multiple reasons.

In order to identify behaviors, we attempted to isolate metrics corresponding to a particular behavior hypothesized to have an effect on debugging behavior. For example, one hypothesis was that students who spend more time in locate-error phase relative to repair-error phase (i.e., students who spend more time thinking about their code) would debug more efficiently than students who spent a larger share of time actively writing code. We instrumented this behavior with the metric "locate-error ratio" in 6.1.2, and measured the statistical relationship between "time spent in locate-error phase" and the count of debugging cycles across all students for both bowlingScore and integerLetterFrequencies.

6.1 Behavioral features

6.1.1 Debugging cycles

Feature 1: **Debugging cycles**, calculated as the number of times the student enters the run-program phase.

The count of debugging cycles is intended as a measure of debugging effort, reflecting how many times a student goes through the "debugging loop," as defined in Section 5.2.

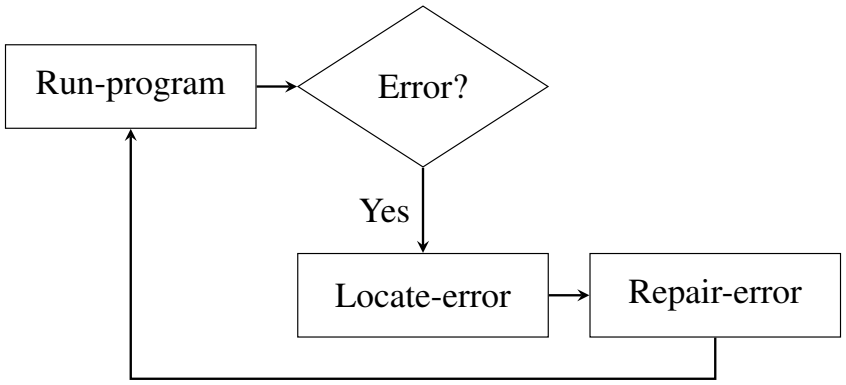

Figure 6.1: **The debugging loop.**

6.1.2 Time spent in locate-error phase

Feature 2: **Locate-error ratio**, calculated as the median ratio between the length of the locate-error phase and the length of the repair-error phase across all cycles in a debugging episode.

The locate-error phase is a critical part of debugging. We measure the amount of time spent in each locate-error phase relative to the time spent in the corresponding repair-error phase. This feature measures the proportion of debugging time spent trying to locate the error versus editing the code.

This feature is meant to measure how much attention students pay to the task of locating errors versus repairing them. We hypothesized that students who spent comparatively longer locating errors would be more likely to correctly fix the error, and would thus finish debugging in fewer cycles.

6.1.3 Print count

Feature 3: **Print count**, calculated as the maximum number of print statements present in the code across all executions.

Since students in this context did not have access to a formal debugger, print statements are a primary tool for observing program state.

We hypothesized that students who used more print statements would be able to debug more efficiently, and would solve problems in fewer debugging cycles, due to collecting more information from each execution of the program from the print statements' output.

6.1.4 Functional edits per repair-error phase

Feature 4: **Functional edits per repair-error phase**, calculated as the total number of behavioral edits made in repair-error phases divided by the number of repair-error phases.

During any given repair-error phase, the edits students make can vary from changing single tokens to rewriting large sections of code. We measure the average number of token-level functional edits students make per repair-error phase.

This feature is meant to measure how focused edits to the code are during repair-error phase. Both the bowlingScore and integerLetterFrequencies exercises could have been debugged by students with only a few tokens changed. We hypothesized that students who changed more tokens than necessary in repair-error phase would take more debugging cycles to finish the exercises.

6.1.5 Summary of Behavioral Features

1. **Debugging cycles**: Number of times the student went through the debugging feedback loop.
2. **Locate-error ratio**: Median ratio between the length of the locate-error phase and the length of the repair-error phase across debugging cycles.
3. **Print count**: Maximum number of print statements in the code across all executions.
4. **Functional edits per repair-error phase**: The average number of edits made during each repair-error phase.

6.2 Method of Analysis

To analyze the data, we first needed to define what constitutes a "struggling" student versus one who is not.

6.2.1 Splitting Groups

We split student debugging episodes into two groups: those with a high number of debugging cycles and those with a low number. Our primary goal is to discover behavioral differences between students who struggled and those who did not.

- **High cycle count** debugging episodes were in the top quartile for the number of debugging cycles for that specific exercise.
- **Low cycle count** debugging episodes were in the bottom three quartiles.
- The threshold number of debugging cycles was 7 for `integerLetterFrequencies`, 8 for `bowlingScore`, and 9 for `moveToBack`.

6.2.2 Statistical Testing

For each of the behavioral features, we performed a Mann-Whitney U test to compare the high-struggle and low-struggle groups. We chose this non-parametric test because our features are not normally distributed within the student population.

6.3 Results

A summary of statistical results is in Table 6.1. Overall, the only effect that was consistently statistically significant and meaningful across both exercises was that of print count. The other two factors had significant effects in the bowlingScore exercise, but not in the integerLetterFrequencies exercise. Because integerLetterFrequencies had a broader sample of the course population, due to

it being mandatory for all students in the course, we believe that locate-error ratio and functional edits per cycle did not have a significant effect on the number of debugging cycles.

Students in the high struggle group actually used significantly more print statements than students in the low struggle group, by a large and statistically significant margin.

Exercise	Measure	Low-Struggle M (n)	High-Struggle M (n)	p-value	Effect
bowlingScore	Locate-Error Ratio	6.28 (131)	6.67 (40)	**0.006**	HS > LS
	Print Statements	0.46 (127)	1.50 (46)	< **0.001**	HS > LS
	Edits per Cycle	7.38 (127)	5.80 (46)	**0.011**	LS > HS
intLetterFreqs	Locate-Error Ratio	4.40 (197)	3.52 (50)	0.265	n.s.
	Print Statements	0.92 (184)	1.58 (67)	**0.003**	HS > LS
	Edits per Cycle	13.46 (184)	10.47 (67)	0.299	n.s.

Table 6.1: **Debugging behavior differences between low-struggle and high-struggle students across two programming exercises.**
Statistical significance assessed via Mann-Whitney U tests. Significant p-values ($p < 0.05$) are shown in bold. HS = High-Struggle, LS = Low-Struggle, n.s. = not significant. intLetterFreqs refers to the integerLetterFrequencies exercise.

6.4 Discussion

None of our hypothesized effects matched up with the results from the process data. The only feature that had a statistically significant relationship with debugging cycles across both exercises was print count, and the effect was actually in the reverse direction of our hypothesis - students who used more print statements debugged less efficiently.

One likely explanation is that students tend to use print statements when they are already struggling - students who are able to debug the problem efficiently did not always need print statements.

As for the other behavioral features (locate-error ratio and functional edits per repair-error phase), there may simply be too much noise in our data. Students were free to complete these debugging exercises anywhere and anytime they wanted to, as long as they completed them within a week of assignment. Any number of environmental and cognitive factors could have affected the amount of time each student spent in each phase. Thus, we hypothesize that collecting data under standardized conditions may be more appropriate for an analysis meant to identify positive and negative behaviors.

Chapter 7

Conclusion

7.1 Summary

- We have collected a large dataset of process data from undergraduate programmers completing debugging exercises. To our knowledge, our dataset is the largest dataset of programming process data ever collected in an educational setting, consisting of more students across more exercises than any other dataset.
- We developed a novel method of extracting information on the cognitive phases of debugging from this process data.
- We observed patterns in the process data that reflect aspects of the behavior of programmers while debugging. These patterns can inform decisions in programming instruction.
- We attempted to use process data to identify efficient and inefficient debugging behaviors, but were unsuccessful. Future work using data collected under laboratory conditions may prove helpful in this aim.
- We provide evidence that process data can be used to evaluate debugging skill.

7.2 Future work - extensions

Our work can be extended and improved in many ways. First, collecting process data from a more advanced course could reveal many new patterns. A comparison of behavior between experts and novices becomes possible when the subject population includes subjects of different skill levels.

There is also more process data that can be collected. One of the chief limitations of the process data we collected is that it is impossible to ascertain what the subject is looking at when they are not actively typing. Integrating some form of eye tracking into the data collection could prove valuable. Alternatively, a modified code editor that only shows one line of code at a time (as used in [Katz and Anderson, 1987]) could be used. This data could enhance our understanding of what exactly students are looking at when in locate-error phase, although it would be more useful for longer exercises than the ones used for this project.

Finally, there are inherent limitations to performing any experiment like this outside of laboratory conditions. Students in the class were free to work on these exercises at any time before the

deadline, and thus were working in a variety of different conditions. If the same experiment were to be repeated under standardized laboratory conditions, we could see different patterns in the data.

7.3 Future work - applications

Process data could be useful for a variety of applications relating to debugging. Below, we have described a few noteworthy possibilities.

7.3.1 Educational interventions

Programming instructors may wish to give their students explicit instructions on debugging. An intervention can take the form of a lecture, a game, or a debugger tool. [Sun et al., 2024] analyzed 18 articles on a variety of different educational interventions and found that many of these interventions are effective in improving debugging skill. Previous research [Liu et al., 2017], [Price et al., 2020] has used process-based evaluations of debugging efficiency as a measure of success for debugging interventions. This research quantifies efficiency as the number of edits or compilations used to fix code, similar to our metric of debugging cycles. Our framework for analyzing process data could help instructors get more detailed feedback on the effect of their interventions.

For example, an instructor could assign students two debugging exercises separated in time, and add a lecture on debugging in between. Differences in debugging behavior between the two exercises could tell the instructor whether students got more efficient due to the intervention. The data could also highlight where those efficiency gains came from - for example, did students get faster in locate-error phase or repair-error phase?

7.3.2 Intelligent tutoring systems

Intelligent tutoring systems (ITS) are "computer programs that are designed to incorporate techniques from the AI community in order to provide tutors which know what they teach, who they teach and how to teach it" [Nwana, 1990]. Several ITSs have been developed and deployed for computer programming instruction [Crow et al., 2018], and [Carter and Blank, 2013] developed an ITS designed to teach students the principles of debugging.

Process data could be useful for developing the next generation of ITSs for debugging and programming in general. For example, designers of ITSs may wish to intervene when a student seems to be struggling to debug a program. Evidence of struggle can be identified in process data, such as a student going through many debugging cycles without making meaningful progress. Process data can be used to create an automatic trigger for hints or other interventions by an ITS.

7.3.3 AI debuggers

A significant amount of research effort has gone into creating systems that can automatically debug computer programs. This line of research goes back to [Katz and Manna, 1975], and continues into the present day. Some recent research has been focused on the potential for large language models (LLMs) to act as automatic debuggers. [Yuan et al., 2025] created an agentic framework

that allows LLMs to call debugger tools in *pdb*, the Python debugger. The authors noted that the agents struggled to use debugging tools in a meaningful way, and performance lagged behind that of human developers more significantly than in general programming tasks. The authors mentioned that training LLMs on human debugging logs as an avenue for future improvement.

Process data of the sort collected in our research is a log of humans debugging. Analysis of process data could yield valuable insight for the design of agentic systems for program debugging. For example, the average length of time spent in each of the cognitive phases (say, locate-error versus repair-error) could inform decisions for computational budget dedicated to each of the subtasks of debugging.

In addition, process data could prove useful for training the underlying LLMs that write and debug code. This method of training LLMs on sequential decision making tasks is largely unexplored as of today.

Appendix A

Exercises

A.1 bowlingScore

The bowlingScore problem statement asks students to write a program that computes the total score of a bowler in a game given a list containing the number of pins knocked down in each throw.

The prompt is as follows:

> Background: When you are bowling, you get 10 frames. In each frame you get 2 throws, where you try to knock down the 10 pins. Your score for that frame is the total number of pins you knocked down in those 2 throws. So if you knock down 3 pins on your first throw, and then 6 more on the second throw, your score in that frame is 3+6, or 9. Your total score is the sum of your score in each frame.
>
> There are some special cases to consider:
>
> If you knock down less than 10 pins on your first throw, but then you knock down the rest of the pins on your second throw, this is called a ”spare”. When you get a spare, your score in that frame also includes your next throw (in the next frame). So if you knock down 3 on your first throw, and 7 on your second throw, that is a spare. So you score 10 plus your next throw. Say your next throw is a 5 (in the next frame). Then your score for the spare is 15. Remember, this throw is counted twice, as it forms part of the score for its frame and the previous frame (the one with the spare).
>
> If you knock down all 10 pins on your first throw in a frame, that is called a ”strike”. In that case, the frame ends. Also, the score for the strike includes the next 2 throws. So if you get a strike, then in the next frame you get 3 on your first throw and 5 on your second throw, then the score for the strike is 10+3+5, or 18. As another example, if you get a strike, followed by another strike, followed by a 3, then the score for the first strike is 10+10+3, or 23.
>
> If you get a spare in the last frame (the 10th frame), you get one more throw, and your score for that last spare includes that last throw. However, there is no 11th frame, even though you got that last throw.
>
> Similarly, if you get a strike in the last frame, you get two more throws, which count towards the 10th frame. Again, though, there is no 11th frame, even though you got both of those last throws.

So we see that the best possible score is if you get a strike in every frame, and then in the last frame you get two extra throws and both of those are strikes, too. In that case, your score is 30 in all 10 frames, so your total score is 300.

With this in mind, write the function bowlingScore(scores) which takes a list of the scores on each throw, and returns the total score for that game.

Remember that when you score a 10, that frame only has 1 throw, except the last frame, as described above.

```
1  def bowlingScore(scores):
2      total = 0
3      i = 0
4      for frame in range(10):
5          if scores[i] == 10:
6              total += 10 + scores[i+2] + scores[i+3]
7              i += 1
8          else:
9              frame_score = scores[i] + scores[i+1]
10             if frame_score == 10:
11                 total += 10 + scores[i+1]
12             else:
13                 total += frame_score
14             i += 2
15     return total
16
17
18 @testFunction
19 def testBowlingScore():
20     assert(bowlingScore([10]*12)==300)
21     assert(bowlingScore([7,2,8,2,10,7,1,8,2,7,3,10,10,5,4,8,2,7])==162)
22     assert(bowlingScore([2,6,2,6,9,1,10,10,10,5,1,4,5,9,0,8,1])==140)
23     assert(bowlingScore([6,4,2,7,8,1,2,4,6,3,10,6,2,1,9,6,4,10,10,10])==137)
24     assert(bowlingScore([8,1,5,3,4,3,0,8,9,0,8,1,3,6,1,8,5,4,7,1])==85)
25
26     # Finally, verify that the function is non-mutating
27     L = [7,2,8,2,10,7,1,8,2,7,3,10,10,5,4,8,2,7]
28     bowlingScore(L)
29     assert(L == [7,2,8,2,10,7,1,8,2,7,3,10,10,5,4,8,2,7])
30
31 def main():
32     testBowlingScore()
33
34 main()
```

Figure A.1: **Buggy starter code for bowlingScore exercise.**
The testBowlingScore function provides test cases for the students' convenience - the final evaluation of correctness uses these test cases as well as additional test cases to check the program. The bugs are on lines 6 and 11.

The starter code for the exercise can be seen in Figure A.1. There are two separate bugs in the code for bowlingScore. On line 6, the index values `i+2` and `i+3` should be changed to `i+1` and `i+2`, respectively, in order to properly score strikes. That is,

```
total += 10 + scores [i+2] + scores [i+3]
```

should become:

```
total += 10 + scores [i+1] + scores [i+2]
```

On line 11, the index value `i+1` should be changed to `i+2` to properly handle spares. The code

```
total += 10 + scores[i+1]
```

should become:

```
total += 10 + scores[i+2]
```

These are the only defects introduced to the original correct version of the code. Some students found alternative paths to fixing the code that involved more changes.

Running the buggy code as given will throw an `IndexError` on the first test case, shown on line 20. This is because the `i+3` index is out of bounds when the value of `i` is 9.

Once the fixes on line 6 are applied, the bug on line 11 will cause the second test case on line 21 to fail with an `AssertionError`. The final total output by the `bowlingScore` function is incorrect because spares are not accounted for properly.

After both bugs are fixed, the code will pass all test cases and the exercise is completed.

Multiple students seemed to have trouble dealing with the `IndexError` thrown when the starter code was executed. Rather than correcting the index itself, students tried to case on whether or not the incorrect index on line 6 would cause the error, and diverting to an edge case branch if it would.

Bibliography

[Ahmadzadeh et al., 2005] Ahmadzadeh, M., Elliman, D., and Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. pages 84–88.

[Alaboudi and Latoza, 2021] Alaboudi, A. and Latoza, T. (2021). Edit - Run Behavior in Programming and Debugging. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10.

[Allwood and Björhag, 1990] Allwood, C. and Björhag, C.-G. (1990). Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies*, 33(6):707–724.

[Alqadi and Maletic, 2017] Alqadi, B. and Maletic, J. (2017). An Empirical Study of Debugging Patterns Among Novices Programmers. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*.

[Becker, 2016] Becker, B. (2016). A new metric to quantify repeated compiler errors for novice programmers. volume 11-13-July-2016, pages 296–301.

[Carter and Blank, 2013] Carter, E. and Blank, G. D. (2013). An intelligent tutoring system to teach debugging. In *International Conference on Artificial Intelligence in Education*, pages 872–875. Springer.

[Carver and Klahr, 1986] Carver, S. M. and Klahr, D. (1986). Assessing children's logo debugging skills with a formal model. *Journal of Educational Computing Research*, 2(4):487–525.

[Crow et al., 2018] Crow, T., Luxton-Reilly, A., and Wuensche, B. (2018). Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian computing education conference*, pages 53–62.

[Edwards et al., 2023] Edwards, J., Hart, K., and Shrestha, R. (2023). Review of CSEDM Data and Introduction of Two Public CS1 Keystroke Datasets. *Journal of Educational Data Mining*, 15(1):1–31.

[Fitzgerald et al., 2008] Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18:116–193.

[Gilmore, 1991] Gilmore, D. (1991). Models of Debugging. page 8.

[Gugerty and Olson, 1986] Gugerty, L. and Olson, G. (1986). Debugging by skilled and novice programmers. pages 171–174.

[Hu et al., 2024] Hu, D., Santiesteban, P., Endres, M., and Weimer, W. (2024). Towards a Cognitive Model of Dynamic Debugging: Does Identifier Construction Matter? *IEEE Transactions on Software Engineering*.

[Hughes and Parkes, 2003] Hughes, J. and Parkes, S. (2003). Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology*, 22(2):127–140.

[Katz and Anderson, 1987] Katz, I. and Anderson, J. (1987). Debugging: An Analysis of Bug-Location Strategies. *Hum. Comput. Interact.*, 3:351–399.

[Katz and Manna, 1975] Katz, S. and Manna, Z. (1975). Towards automatic debugging of programs. In *Proceedings of the International Conference on Reliable Software*, page 143–155, New York, NY, USA. Association for Computing Machinery.

[Kessler and Anderson, 1986] Kessler, C. and Anderson, J. R. (1986). A Model of Novice Debugging in LISP. In Soloway, E. and Iyengar, S., editors, *Empirical Studies in Programmers*. Ablex, Norwood, NJ.

[Lin et al., 2016] Lin, Y.-T., Wu, C.-C., Hou, T.-Y., Lin, Y.-C., Yang, F.-Y., and Chang, C.-H. (2016). Tracking students' cognitive processes during program debugging: An eye-movement approach. *IEEE Transactions on Education*, 59:175–186.

[Liu and Paquette, 2023] Liu, Q. and Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies. *LAK23: 13th International Learning Analytics and Knowledge Conference*.

[Liu and Paquette, 2024] Liu, Q. and Paquette, L. (2024). Applying Sequence Analysis to Understand the Debugging Process of Novice Programmers. volume 3796.

[Liu et al., 2017] Liu, Z., Zhi, R., Hicks, A., and Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, 27(1):1–29.

[Myers, 1978] Myers, G. J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Commun. ACM*, 21(9):760–768. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[Nwana, 1990] Nwana, H. S. (1990). Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 4(4):251–277.

[Price et al., 2020] Price, T. W., Marwan, S., Winters, M., and Williams, J. J. (2020). An evaluation of data-driven programming hints in a classroom setting. In Bittencourt, I. I., Cukurova, M., Muldner, K., Luckin, R., and Millán, E., editors, *Artificial Intelligence in Education*, pages 246–251, Cham. Springer International Publishing.

[Rasmussen and Jensen, 1974] Rasmussen, J. and Jensen, A. (1974). Mental procedures in real-life tasks: A case study of electronic trouble shooting. *Ergonomics*, 17(3):293–307.

[Sun et al., 2024] Sun, C., Yang, S., and Becker, B. (2024). Debugging in Computational Thinking: A Meta-analysis on the Effects of Interventions on Debugging Skills. *Journal of Educational Computing Research*, 62:1087–1121.

[Timmerman et al., 1993] Timmerman, M., Gielen, F., and Lambrix, P. (1993). A knowledge-based approach for the debugging of real-time multiprocessor systems. In *[1993] Proceedings of the IEEE Workshop on Real-Time Applications*, pages 23–28.

[Vessey, 1984] Vessey, I. (1984). Expertise in Debugging Computer Programs: A Process Analysis. *Int. J. Man Mach. Stud.*, 23:459–494.

[Vessey, 1985] Vessey, I. (1985). Expertise in Debugging Computer Programs: Situation-Based versus Model-Based Problem Solving. page 18.

[Vessey, 1986] Vessey, I. (1986). Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, 16:621–637.

[Whalley et al., 2023] Whalley, J., Settle, A., and Luxton-Reilly, A. (2023). A Think-Aloud Study of Novice Debugging. *ACM Transactions on Computing Education*, 23:1–38.

[Yuan et al., 2025] Yuan, X., Moss, M. M., Feghali, C. E., Singh, C., Moldavskaya, D., MacPhee, D., Caccia, L., Pereira, M., Kim, M., Sordoni, A., and Côté, M.-A. (2025). debug-gym: A text-based environment for interactive debugging.