NORTHWESTERN UNIVERSITY

Investigating student self-perceptions during the programming process

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science and Learning Sciences

By
Jamie Sarah Gorson

EVANSTON, IL

August 2022

©Copyright by Jamie Sarah Gorson

2022 All Rights Reserved

ABSTRACT

While there is high demand for university computer science (CS) courses, students often struggle when learning to program. Prior work has identified that student perceptions of their programming ability may contribute to these challenges. For example, studies show that students often perceive that they do not belong, are not capable of succeeding, or are performing poorly in CS. In this dissertation, I investigated the student experience working on programming problems and how these experiences influenced student perceptions of their programming ability. While prior studies have investigated the application of mindset and self-efficacy theories in CS, we do not currently know much about how students make self-evaluations while working on programming problems and how their perceptions of programming intelligence impacts their programming experiences. To address this gap, I first explored how students talk about programming intelligence. In doing so, I identified the criteria that students used to evaluate their programming ability. Since many of these criteria relate to particular moments in the programming session, I next investigated if students feel like they are performing poorly when they encounter those moments. Thus, I identified a list of moments during a programming session where some students make negative self-assessments. I then conducted follow-up analyses to investigate two potential explanations for why students feel poorly during those moments. To build a tool for automatically identifying the moments that cause students to feel like they are performing poorly, I developed a methodology for creating automated detection systems based on student perceptions. Finally, I explored the events that trigger students to have emotional reactions while programming in order to better understand how students experience working on programming problems. From my dissertation research, we have a better understanding of how students experience the programming process and how students develop their perceptions of programming ability, particularly while working on programming problems. From this work, I make four types of contributions: practical, conceptual, methodological, and technological. Specifically, I contribute the start to a new framework for understanding how students evaluate themselves while programming, providing new insights and

tools to understand the student programming experience. I provide a list of moments when students negatively self-assess during programming and show that students who report to negatively self-assess at more of these moments tend to have lower self-efficacy. In addition, I contribute new methods that can be used to further our understanding of student perceptions of their programming experiences. Finally, I further our understanding of events that cause novices to experience emotions while programming and demonstrate that patterns from physiological data sources can help to represent student experience. These contributions help instructors of CS programs, CS education researchers, and designers of interventions to better support university students' self-efficacy, persistence, and performance as they learn to program.

ACKNOWLEDGEMENTS

In the process of reflecting on my six-year graduate school journey, there are many people who played important roles that I wish to acknowledge and thank.

First, my adviser, Dr. Nell O'Rourke. The passion and care that you put into mentoring students comes through on a daily basis. Throughout my graduate school journey, you have taught me about more than just research. Particularly, I appreciate all of you mentorship on my writing, whether it was feedback on many practice papers or co-writing conference submissions, you have taught me to be a much more eloquent and articulate writer. Thank you for being the model, mentor and friend that I have looked up to for the past 5 years and I only hope that we can stay close for many years to follow.

Thank you to my committee members, Dr. Michael Horn, Dr. Marcelo Worsely, and Dr. Mark Guzdial. You each deeply inspire me with your work and the innovation that you bring to the research community. Thank you for supporting my research with feedback, insights, and encouragement.

Thank you to the other innovative leaders of the Delta Lab: Dr. Matt Easterday, Dr. Haoqi Zhang, and Dr. Liz Gerber. Your mentorship, feedback, and leadership pushes me to be a better researcher and community member.

Thank you for all of the feedback, encouragement and collaboration from my colleagues in Nell's SIG: Garrett Hedman, Harrison Kwik, Gobi Dasu, Nick LaGrassa, and Dr. Katie Cunningham, Thank you to my supportive and thoughtful peers in the Delta Lab: Dr, Emily Harburg, Dr. Dan Rees Lewis, Dr. Julie Hui, Dr. Eureka Foong, Dr. Yongsung Kim, Dr. Spencer Carlson, Ryan Louie, Kristine Lu, Kapil Garg, Gus Umbelino, and Evey Huang. I want to especially thank Leesha Maliakal Shah for being with me from day 1 of graduate school all the way until the end. I looked forward to our weekly meetings and am grateful for your encouragement, feedback, and friendship. Thank you to the undergraduate students who have worked with me in the past few years, Cindy Hsinyu Hu, Elise Lee, Ava Marie Robinson, and Yaurie Hwang. Your contributions

to my work have helped me to do great research.

Thank you to the Northwestern Staff, both in the Computer Science and Learning Science departments who have cared about me throughout my time in graduate school. Thank you to Dr. Uri Wilensky and Dr. Mike Horn for creating and leading the CS+LS program. I could not imagine a program that better fit my interests. I appreciate your efforts in putting the program together and creating such a fantastic community around it.

Thank you to the welcoming community in EngEDU at Google, including Dr. Phil Nova, Diana Gage, Maia Deutsch, my Grasshopper teammates, Nicole Le, Darla Sharp, and many more. I especially want to thank Dr. Kyle Jennings and Dr. Chris Stephenson for advocating for me to join Google. I am grateful for your faith in me and support in achieving my goals. I look forward to returning and continuing to apply my research skills and experiences with such an impactful group.

Thank you the Chicago communities that I have joined over the last six years. From the fitness world to the Jewish community, the friendships and encouragement that I have found from being a part of these communities have kept me going.

Thank you to the National Science Foundation Graduate Research Program Fellowship for funding my research and believing in me as a rising researcher.

Most importantly thanks to my family. To my siblings, Michelle and Eric, thank you for being my biggest cheerleaders, friends, support system, and adventure buddies in the last few years. While the pandemic turned our world upside down, I will forever be grateful for the opportunity that we had to live together as adults and that we continue to be as close as always. Kaylee - thank you for seamlessly joining our family and being with us through the ups and downs of the last few years. To my in-laws and brother-in-law, Julie, Howard and Steven. Thank you for welcoming me into your family with open arms. Your excitement for me finishing the PhD is unrivaled and I have loved getting to share my journey with you each step of the way. To my Mom, Phyllis, thank you for supporting my decisions and choices throughout my whole life, no matter which direction they went. Your work ethic and drive has inspired me to be who I am. Thank you for

reading every one of my papers, no matter how confusing they were or late it was at night. Finally, my husband Paul. I never expected to meet someone while intensely preparing for my qualifying exams, but there you were. You push me to be a better person, writer, friend, scholar and wife. Thank you for dealing with my stress that comes from paper deadlines, my defensiveness when getting presentation feedback, and the crazy lifestyle a PhD student lives. I could not imagine going through this journey without you by my side. I love you all.

In loving memory of my dad, Murray Gorson. Thank you for inspiring my drive to explore new technologies and my passion for improving education. You were the biggest supporter and advocate of my education, from elementary school projects to integrated middle school programs and college capstones. Your excitement for interdisciplinary and hands-on learning is the origin of my now lifelong goals and aspirations. You are the reason that I followed this path; I would not be here today without your love and inspiration.

TABLE OF CONTENTS

Acknowledgments 5

Dedication 8

List of Tables 14

List of Figures 15

Chapter 1: Introduction 17

1.1 Problem description 17

1.2 Research approach 19

1.3 Research objectives of the studies in my dissertation 22

1.4 Contributions of the research 24

Chapter 2: Background 27

2.1 Mindset theory 28

2.2 Self-efficacy theory 29

2.3 Self-assessments 32

2.4 Gap in literature 33

Chapter 3: Study 1: How do students talk about programming intelligence? 34

3.1 Problem and background 34

3.2 Interview study . . . 35
3.2.1 Methods . . . 35
3.2.2 Data analysis . . . 37
3.2.3 Mindset findings . . . 38
3.2.4 Self-assessment criteria findings . . . 42
3.2.5 Discussion . . . 44
3.3 Survey study . . . 45
3.3.1 Participants and setting . . . 46
3.3.2 Open-ended survey question . . . 46
3.3.3 Likert scale survey questions . . . 48
3.4 Conclusion . . . 51

Chapter 4: Study 2: Why do CS1 students think they're bad at programming? Investigating self-efficacy and self-assessments at three universities . . . 53

4.1 Problem and background . . . 53
4.2 Methods . . . 54
4.2.1 Survey design . . . 55
4.2.2 Participants . . . 59
4.2.3 Survey procedure . . . 60
4.2.4 Follow-up interview procedure . . . 61
4.2.5 Findings . . . 61
4.2.6 Students from all three universities reported negative self-assessments . . . 62
4.2.7 Students understood and related to the vignettes . . . 63
4.2.8 Students who self-assess more frequently have lower self-efficacy . . . 67

4.2.9 Perceptions of professional programmers may influence self-assessment moments 69

4.2.10 Students evaluate themselves more critically than they evaluate others . . . 72

4.2.11 Self-critical bias is stronger when the student or the vignette character is female 74

4.3 Conclusion 75

Chapter 5: Study 3: An approach for detecting student perceptions of the programming experience from interaction log data 78

5.1 Problem and background 78

5.2 Use of interaction log data in CS Education 79

5.3 Retrospective-enabled perception recognition 81

5.3.1 Data collection tools 81

5.3.2 Phase 1: Retrospective interviews 81

5.3.3 Phase 2: Qualitative analysis 83

5.3.4 Phase 3: Codebook verification 84

5.3.5 Phase 4: Implementation of the detection system 84

5.4 Evaluation of the system 86

5.4.1 Methods 86

5.4.2 Findings 87

5.5 Conclusion 89

Chapter 6: Study 4: Using electrodermal activity measurements to understand novice programmer emotions 91

6.1 Problem 91

6.2 Background 93

6.2.1 Emotions . . . 93

6.2.2 Existing methodologies for identifying triggers of emotions during novice programming . . . 94

6.2.3 Physiological data analytics - electrodermal activity . . . 95

6.3 Method . . . 98

6.3.1 Participants & setting . . . 98

6.3.2 Study procedure . . . 98

6.3.3 SCR detection . . . 100

6.3.4 Identification of triggers of emotions . . . 101

6.3.5 Analysis of EDA data with respect to student experiences . . . 101

6.4 Findings . . . 102

6.4.1 Events that trigger student emotions . . . 102

6.4.2 Emotional experiences reflected in EDA data . . . 112

6.5 Discussion . . . 119

6.6 Limitations . . . 121

6.7 Conclusion . . . 122

Chapter 7: Dissertation Conclusion . . . 124

7.1 Future work . . . 128

References . . . 142

Chapter A: Qualitative Codebook for Researchers to Identify Moments of Potential Self-Assessment . . . 143

A.1 Guide for labeling using resources moments . . . 143

A.1.1 Definitions of terms and concepts 144

A.1.2 Using resource for syntax 148

A.1.3 Using resource for approach 153

A.1.4 Using resources in general 157

A.2 Getting simple errors 159

A.3 Getting Java error 161

A.4 Struggling with error 162

A.5 Writing a plan 165

A.6 Stopping to think after starting implementation 168

A.7 Changing approach 171

A.8 Unexpected runtime behavior 174

A.9 Glossary of terms and instructions 177

A.9.1 Term: Main Function 177

A.9.2 Term: Implementation 177

A.9.3 Term: Error Cycle 177

A.9.4 Term: Meaningful Code 178

LIST OF TABLES

3.1 Qualitative codebook for analysis in Study 1 38

3.2 Self-assessment criteria codebook 43

4.1 The thirteen self-assessment moments and the vignettes that we included on the survey. 56

4.2 Demographic data from survey participants for each university in percentages . . . 60

4.3 Results of the statistical analysis of the self-assessment moment vignette-style survey questions 64

5.1 Negative self-assessment moments detected by the expert system 82

5.2 Results from the evaluation of the detection system 88

6.1 Events that triggered positive emotions during the programming session. 103

6.2 Events that triggered negative emotions during the programming session. 104

LIST OF FIGURES

3.1 Graph of participant mindset talk relative to associated behaviors 39

3.2 Variation of participant responses to self-assessment criteria survey questions . . . 47

3.3 Histogram of responses to the survey question: *Being able to explain your program is an indication of programming intelligence* 49

3.4 Histogram of responses to the survey question: *Someone is more intelligent at programming if they do an assignment on their own, rather than getting help to solve it* 50

4.1 Histograms of the responses to each of the self-assessment vignette-style survey questions 62

4.2 Graph showing the self-assessment compound score versus the self-efficacy compound score by university 69

4.3 Graph showing the participant self-critical biases for each moment 73

5.1 Timeline graph demonstrating the self-assessment moments that occurred in a participant interview 83

6.1 Phasic and tonic activity of EDA signal 95

6.2 The Empatica E4 wristband [140]. 96

6.3 Results from skin conductance response (SCR) detection for P12. SCRs are marked by dotted vertical lines. EDA level is displayed in blue. 100

6.4 EDA graphs of participants demonstrating the ”Cruise Control” pattern, which begins on each graph at the Marker X. 113

6.5 EDA graphs demonstrating high emotion sections, indicated by the shading. 115

6.6 EDA graphs demonstrating participants who experienced few negative emotions while working on the programming problem. 118

CHAPTER 1
INTRODUCTION

1.1 Problem description

University computer science (CS) departments are experiencing high demand for CS courses as college students increasingly desire programming skills. This growth coincides with an increased need for technology talent in industry as software development jobs are growing much faster than the average occupation [1] and employees with programming skills tend to receive higher wages [2, 3]. Programming is being introduced at younger ages and has been established as a necessary literacy for all learners, driving more people to the field [4]. However, many students in introductory computer sciences courses (CS1) at the university level struggle to learn to program [5, 6].

While there are many possible sources causing students to struggle, prior studies indicate that student perceptions of their programming ability strongly contribute to this issue. Specifically, studies show that students often perceive that they do not belong [7, 8, 9], are not capable of succeeding [10, 11, 12], or are performing poorly in CS [6, 13, 14, 15]. For example, Kinnunen and Simon found that, at times, students perceived that they performed poorly even when they successfully completed a programming problem. This misalignment often occurred when the students' programming process did not match their expectations or perceptions of good programming performance [14]. These perceptions may also factor into the emotions that arise while programming, as students experience a roller-coaster of emotions while working on programming problems [6].

In addition to impacting student experiences, these perceptions of the programming process influence student assessments of their programming ability, which factor into their decisions to persist in CS [15]. Self-efficacy is the belief in one's ability to accomplish a task or achieve mastery in a specific domain [16, 17, 18]. Studies have shown that self-efficacy has a direct impact

on student learning outcomes [19] and often correlates with student performance in CS courses [20, 21, 22], thus suggesting the need for further research into student perceptions. Prior work indicates that student perceptions of their programming experiences may correlate with students' self-efficacy [23, 24, 14, 7], aligning with self-efficacy theory [23]. For example, one study found a direct relationship between student sense of belonging in programming with their self-efficacy [7]. However, few studies have investigated how students perceive their programming sessions and how these perspectives influence student self-efficacy. Additionally, studies have shown that through an introductory programming course students increasingly believe that their programming intelligence is fixed and cannot be changed. We do not currently know what causes this belief and how it impacts student programming sessions.

The history of computing education may contribute to the origins of why students have negative perceptions of their programming ability. Prior to the existence of CS courses at universities, businesses trained their programmers in-house. Since there was a high failure rate amongst those in these roles, businesses needed a way to determine if potential employees would succeed in programming-related jobs [25]. Almost 80% of businesses implemented aptitude tests for evaluating employee's potential to succeed in CS [26]. Despite their widespread implementation, aptitude tests did not work as well as expected; there was only a modest correlation between an individual's score on the test and their assessed programming skill [27]. Unfortunately, even if the tests were unsuccessful, by using an aptitude test as an evaluation of success in programming, the tests promote the belief that an innate trait or aptitude is necessary to succeed in CS. Thus, these historic tests may contribute to why many programming students today see programming skills as innate.

We do not have a strong understanding of student perceptions of the programming process and how those perceptions influence broader constructs like motivation and self-efficacy. For example, while performance on tasks is one of the main information sources to self-efficacy [23], we do not know much about how student experiences working on programming problems influences these evaluations. The goal of my dissertation is to deepen our understanding of how student programming experiences influence their perceptions of their programming ability, focusing on

how and when students make self-judgements during programming. My work prioritizes student perceptions, ensuring that I study the programming process from their point of view and not from expert-defined or theory driven concepts. I take this angle because I want to encourage the design of interventions that account for student experiences when addressing the challenges of novice programmers.

1.2 Research approach

As understanding student perspectives is complex, there are a number of challenges to conducting studies with this research goal. First, the constructs are nebulous. Emotions, student reactions, and criteria for defining ability are not concrete as they exist in students' minds. Emotions blend together, student reactions do not always fit into categories, and students do not have an explicit internal rubric for evaluating themselves. Thus, the topics that we are studying are hard to define and differentiate. Second, students often have a hard time expressing themselves and talking about their experiences. These constructs are personal and talking about them is uncomfortable; also students may not have thought about these concepts in the past, so they do not know how to respond when the questions are asked. Finally, these moments of interest happen throughout students' experiences working on their programming problems, which can occur at different times during a programming session. This makes it challenging to capture data specifically when these moments occur.

Surveys and qualitative interviews are two of the established methodologies used to investigate student perceptions. While we can learn a lot about student experiences using these methods, additional considerations and strategies may be necessary to address the challenges described above. For examples, surveys are restricted to what students self-report and force rigid categories. Qualitative interviews address some of those concerns by allowing a discussion between the interviewer and the student, but they are restricted by what students can recall and are willing to share during the interview. In my dissertation studies, I address some of these challenges and limitations in my research approach. Through designing the methods of the four studies, a set of guiding principles

emerged. These principles aim to improve our ability to collect precise and descriptive data. As I progress through the dissertation, I move away from the more traditional research approaches and more strongly incorporate these guiding principles into the study design. In the last two studies, I make methodological contributions, documenting new methods for understanding the student programming experience that follow this set of principles.

The guiding principles are:

- *Mix qualitative approaches with sensor or logged data sources.* By mixing data logged from programming episodes with qualitative methods, we can use the precise details captured from the concrete sources, like keystrokes or electrodermal activity (EDA) data, in combination with detailed perspectives from students. Additionally, sensor and logged data sources are measured and do not rely on student reports. Thus, they provide an objective data source to ground the more descriptive and subjective details gathered from students' reports. By combining the data sources, we benefit from the advantages of both approaches, incorporating the nuances and details that arise from qualitative data with the precision in the timing and specificity of information from quantitative data sources.

- *Design studies that enhance student ability to recall relevant data.* When the main data source for a study relies on students to describe their experiences, it is necessary to design methods that aid students in providing accurate recollections. This is especially important when asking students to recall sub-cognitive events, like emotions and self-evaluations, as they are less present in students' minds. Without this consideration, students may not know what to report or may make something up in order to answer the researchers' question [28, 29]. Thus, I design study methods to increase the precision of student recall. For example, I situate programming episodes as closely as possible to interviews and provide information sources to probe students' memories.

- *Consider participant state of mind.* When designing study protocols, whether it is an interview or survey, I consider students' state of mind when they are answering questions. There

are a few different factors that contribute to the participant state of mind. First, it is important to consider how the experiences and emotions that arise for students directly before the research session may impact their responses and behaviors. For example, to accurately investigate if students have a growth or fixed mindset, it is important that they recently encountered a difficult task in order to reveal their reaction to challenge and viewpoint on the malleability of intelligence. Thus we ask all students to work on a challenging problem before the interview and survey. Second, when taking a survey, it is important that students are in an environment where they can give their full attention. Finally, the context and environment that surround the student may also impact how they respond to questions.

- *Create authentic programming episodes.* Since my research is done using lab studies, I emphasize increasing the authenticity of the programming environment. When students participate in programming sessions during a study, I design the environment to create as natural of a programming experience as possible. This increases the likelihood that students experience the same emotions, make the same self-evaluations, and use the same programming strategies that they would outside of the laboratory. The strategies that I use for creating authentic programming environments include: I do not interrupt them throughout a session, they program using their own computers, and I provide similar programming problems to their assignments in their coursework.

With these guiding principles, I designed four studies that helped us to learn about, identify, and detect the moments that students make negative self-assessments through a programming episode and identify the events throughout a programming session that cause students to experience emotions. The studies were able to produce accurate and precise insight into student experiences, suggesting the success of using these principles. The methods for each chapter are different, but they all follow this overall approach. By using different methods in each chapter, I demonstrate how each method can provide new insights and perspectives to understanding the student programming experience. Additionally, in two of the chapters I make methodological contributions; I designed a method for developing detection systems based on student perceptions, and I developed a method-

ology for utilizing physiological data to enhance student recall during retrospective interviews of programming sessions. These two chapters provide example methods that future researchers can follow to incorporate this approach into their future studies.

Due to the qualitative nature and human elements to this research, I, as the researcher, am an instrument in the data collection and the analysis. It important to recognize that despite any effort to be unbiased, my background, personality and viewpoint will influence both the data collection and analysis. For example, during interviews, I develop rapport with the participants to help them feel comfortable sharing, and while I follow interview protocols, the follow-up questions and phrasings are decided during the interview. Similarly in the analysis, it is possible that my viewpoints influence the outcomes, as I see the data through my perspective of the world. To address this bias, I utilize strict protocols and test the reliability of qualitative analysis with other researchers through inter-rater reliability. No matter the method for addressing the bias, since I as a researcher am an instrument, I am likely to make an impact on the outcomes.

1.3 Research objectives of the studies in my dissertation

There are four studies that comprise my dissertation work. They are:

- **Study 1:** My collaborator and I began this line of research by exploring how students perceive and evaluate their programming intelligence, which I detail in Chapter 3 [13]. Building on mindset theory, we were specifically interested in student perspectives on if programming intelligence can change or if it is innate. We interviewed introductory CS students about the nature of programming intelligence directly after they worked on a challenging programming problem. From the interview data, we investigated the application of growth mindset theory in CS. In doing so, we noticed that students evaluated their programming ability frequently, using surprising criteria. Thus, we conducted a survey study to identify a list of criteria that students use to evaluate programming intelligence, which we call self-assessment criteria. Many of these criteria were surprising because they did not align with how professional programmers and instructors would evaluate programming ability.

- **Study 2:** Since many of the self-assessment criteria referenced particular moments in the programming process, we next investigated if students self-assess when those moments arise. We conducted a survey study with students from three universities, reaching a large number of students, which we present in Chapter 4 [30, 31]. Our results showed that some students negatively self-assess when they encounter these moments and students who more frequently make negative self-assessments tend to have lower self-efficacy. Additionally, we explored three potential explanations as to why students self-assess in these moments. These explanations were related to their: self-critical bias (evaluating one-self more harshly than others), gender, and perceptions of professional programmers. We found that some students are more likely to negatively self-assess at moments that they believe professional programmers do not encounter and some students have a self-critical bias, evaluating themselves more harshly than others. We additionally investigated the relationship between self-critical bias, gender and sense of belonging.

- **Study 3:** While we now have identified moments of interest during a programming session, in order to further study them in context or intervene when they arise, we need to be able to automatically identify when they occur. To address this, we developed a new methodology which we call *retrospective-enabled perception recognition* to build a detection system, presented in Chapter 5 [32]. To build the detection system, we gathered student-labels of moments of programming sessions and analyzed the patterns from student interaction behaviors to identify indicators of the self-assessment moments. This allowed us to develop a codebook and an automated tool to identify the self-assessment moments in student programming sessions. In an evaluation comparing our tool to researcher-identified moments, our tool showed promising results, with F1 scores ranging from 66% to 98%, suggesting this approach as a valid method for future studies.

- **Study 4:** While in our first three studies we learned that students report to negatively self-assess in moments during the programming process, and prior work indicates that students

have strong emotions during their programming sessions, we do not know how students experience those self-assessment moments. Additionally, few studies have been able to identify the specific events that cause emotions during the programming process. To address this gap, we investigated the moments that cause students to experience emotions, which I present in Chapter 6 [33]. We asked novice programmers to wear electrodermal activity (EDA) sensors during a programming session, which we used in a retrospective interview to cue their recall of the events that caused them to experience emotions. Additionally, we visually analyzed their EDA data across the programming session to find patterns in the EDA data that relate to student experiences. We identified 23 events that prompt students to experience emotional reactions while programming (9 positive and 14 negative). Additionally, we found 3 patterns in the EDA data that relate to student perspectives on their programming experience.

1.4 Contributions of the research

In Study 1, we contribute empirical findings on the complexity of student perceptions of programming intelligence. We show that students do not always have a clear mindset towards the malleability of programming intelligence and students do not necessarily behave as expected based on mindset theory. We also identified that students assess their programming ability frequently. We contribute a novel list of criteria that students use to evaluate their programming ability. Our findings from this study suggest that student mindsets towards their programming intelligence may interact with their other perceptions of ability and thus mindsets can not be the only theory of intelligence that we use in designing curricular and technological interventions for CS1 students.

In Study 2, we contribute evidence that many students negatively self-assess when they encounter particular moments in the programming process, even though those moments are natural parts of professional practice. These moments are particularly influential because we found that students who negatively self-assess more frequently at these moments tend to have lower self-efficacy. We also contribute two explanations for why students negatively self-assess when these moments occur. We found that many students have inaccurate perceptions of professional pro-

grammers and we found a correlation between students' perceptions of professional programmers and if they negatively self-assess for some of the moments. We also found that many students have a self-critical bias, evaluating themselves more harshly than others. Female students tended to have a stronger self-critical bias than males, and students in general tended to be more self-critical when evaluating themselves relative to their evaluations of female students compared to male students. From these studies, we contribute a deeper understanding of the explanations of the self-assessment moments.

In Study 3, we contribute a new method for designing tools to automatically identify moments based on student perceptions. Our tool for identifying the self-assessment moments from interaction-log data of a programming session demonstrates that student perceptions can be identified based on interactions with the computer. While prior studies have used log data for tasks like predicting student performance and identifying the progress in student solution states, none have been successful in identifying moments based on student perceptions. Thus this novel methodological approach enables researchers to develop detection tools that can identify moments based on student perceptions of the programming process.

In Study 4, we contribute a list of 23 moments that cause novice programmers to experience either a positive or a negative emotional reaction while working on a programming problem. Through visual inspection of EDA data, we identified patterns across the programming session that coordinated with students' programming experiences, suggesting the usefulness of EDA data for computing education researchers who have the goal of understanding programming student emotions. We contribute the methodology for incorporating EDA data with retrospective interviews. Our findings suggest that this approach can support researchers in understanding student emotional reactions during many types of coding activities, in comparing student experiences on programming problems, and in isolating particularly interesting segments of a programming session.

Across my dissertation research, I make practical, conceptual, methodological, and technological contributions. Specifically, with the self-assessment criteria, my colleague and I contribute a completely new perspective from which to understand the student programming experience and

how students perceive and assess programming ability. We are the first to explore how students evaluate their programming intelligence and lay out the specific criteria that students use to evaluate themselves. With this new framework, we were then able to study these moments as they arise. These findings help us to build a stronger conceptual model of the student programming experience and additionally provide practical directions for instructors of programming courses. Practically, our results inform instructors on how students interpret their experiences during the programming process, which could help instructors provide feedback to students that increases their motivation. I also contribute tools and methodologies to aid further research of student programming experiences and the design of technological interventions. This includes a tool for detecting moments of self-assessment and a method for studying moments when students experience emotions while programming. These contributions pave the way for researchers to design studies to better understand how students experience programming problems and build technological interventions to improve these experiences.

One important consideration for this type of research is that these findings may be different if we conducted this research in a different socio-cultural context. While some of the work does compare students from different types of universities, these studies all work with a specific group of people, mostly introductory computer science students in University contexts. Thus, when generalizing these contributions to other contexts, for example K-12, vocational schools, or corporate training, there may be differences in how these students experience programming episodes and consider their programming intelligence.

Moving forward as a field, computing education researchers should continue to study student experiences while programming, as it is crucial to student learning and important to support the cognitive aspects of computing education. In order to continue to address the challenges in understanding student perspectives of themselves while programming, I suggest that future researchers follow, and continue to build upon, the guiding principles presented in the research approach as well as the methodological approaches when designing studies that investigate student programming experiences.

CHAPTER 2
BACKGROUND

The CS learning environment is a very complex space, with many potential factors contributing to the student experience. First, as enrollment in CS programs continues to rise, class sizes are often very large; high student to faculty ratios result in little direct interactions with faculty. Thus, large classes tend to have higher failure rates [34]. Second, students often enter CS1 with a variety of prior programming experiences, which can be problematic because students who are new to programming often struggle when grouped in classes with more experienced peers [35]. Additionally, CS1 students feel pressure to decide if they should major in the domain [15], as many university CS students are trying computing for the first time and are in the process of choosing their college trajectory. Also, computer science development environments are inundated with negative performance feedback, for example error messages and failed test cases. Negative performance feedback often promotes the fixed mindset and lowers student self-efficacy, both of which I will discuss shortly [36, 23, 37]. These challenges are often amplified for women and students of color, who are underrepresented in computer science. These students faced strong stereotypes of who belongs as a computer scientist [38] and often drop out of the major at higher rates [39, 40]. Thus, while programming skills are increasingly important for 21st century learners, many students struggle in introductory computer science (CS) courses [5, 6].

Recent studies suggest that these challenges may be exacerbated by students' self-perceptions; students often believe that they do not belong [7, 8, 9], are not capable of succeeding [10, 11, 12], or are performing poorly in CS [13, 14, 15, 6]. In my dissertation, I study how these perceptions interact with student experiences working on programming problems. In the following sections, I both review related theories, mostly derived from psychology literature, and discuss how those theories are currently understood in CS.

2.1 Mindset theory

Student mindset towards the malleability of intelligence is a widely studied subject. Specifically, research has established that students with a growth mindset believe that intelligence is malleable and can grow through effort and practice, while students with a fixed mindset believe that intelligence is an unchangeable attribute and there is a limit to each person's potential growth [41, 42, 43]. Psychology studies have demonstrated that student beliefs about the malleability of intelligence can have a strong impact on their motivation, reaction to challenge, and academic performance [44, 45]. Students with the growth mindset tend to value learning over performance, and are more likely to persist when challenged, while students with the fixed mindset view challenges as tests of their intelligence, and see mistakes as evidence of low ability [44, 36]. These studies indicate that students with a growth mindset tend to show greater improvements in course performance in comparison to students with a fixed mindset. Specifically, a study analyzing student grade changes over two years in middle school found that students with a growth mindset tended to have increasing grades across the two years while students with a fixed mindset tended to have static or a downward trajectory in their grades [44].

Studies exploring mindset theory in CS have found that the fixed mindset is particularly prevalent across the domain. Specifically, multiple studies have recorded that student mindsets become significantly more fixed through their first programming course [12, 11, 10], indicating that the increase in fixed mindset may stem from CS1 learning environments.

Fortunately, psychologists have designed a number of interventions that demonstrate that student mindset towards intelligence can be influenced [44, 46, 47]. One type of intervention involves directly teaching about the malleability of intelligence. For example, in Aronson et al.'s intervention, students learned about the growth mindset and then taught younger students about it, which they found had a significant positive impact on participant mindset and GPA [46]. Another intervention type involves providing feedback to students at opportune moments. For example, a study showed that students who receive feedback about their process instead of their performance

through an online learning game are more likely to persist through challenges [48]. This type of intervention has been shown to be effective in mediating student perceptions [48, 49] and can scale to large student populations.

A number of CS education researchers have attempted mindset interventions in CS, and have had varied success in impacting student persistence and performance behaviors compared to other domains [12, 50]. For example, Cutts et al. designed a direct teaching intervention in which tutors gave mindset lessons and messages to CS students [12]. While the intervention successfully changed student mindsets on surveys, it did not change course performance. Similarly, Simon et al. replicated Aronson et al.'s successful intervention [46], in which they taught students about mindset theory through a 'saying is believing' exercise, but found no significant effect on students' mindsets in the domain of programming [50]. The mixed intervention results demonstrate that there is a gap in our understanding of how to apply mindset theory in the CS education domain and how a students' mindset may impact their experiences and behaviors in a programming session. However, mindset is not the only theory that can help provide insight into student perspectives on their intelligence. Thus, we next discuss other theories that may help us interpret how student perceptions of programming intelligence may impact their behavior and motivations in CS.

2.2 Self-efficacy theory

While we do not yet fully understand the relationship between self-efficacy and mindset, both theories provide lenses for understanding student perspectives on programming intelligence that may help address the gap in our understanding of how students perceive their experiences programming. Self-efficacy is a widely used construct to represent an individual's judgment of their ability to execute tasks or achieve mastery in a particular domain [16]. Self-efficacy is specific to a domain as students can have a different self-efficacy in different domains. Self-efficacy is based on four principal sources of information: (1) enactive attainments, or the results of performing tasks related to mastering a subject, like passing a test, (2) vicarious experiences observing others perform subject-related tasks, (3) verbal persuasion from others, like words of encouragement, and

(4) physiological states, like stress [23]. Enactive attainments, or the results of performing tasks related to subject mastery, are the most influential of these sources because the information comes from the performance of tasks that contribute to mastering the domain. Self-assessments, which are how people interpret their performance on a task, determine their enactive attainments and thus inform their self-efficacy evaluations. I discuss self-assessments specifically in the next section.

Students with higher self-efficacy are more likely to persist through challenges and be resilient through their long-term goals [16, 51]. Studies have shown that self-efficacy can impact career choice and persistence through college majors [52, 53, 54, 55]. For example, Betz & Hackett found that college students with higher self-efficacy in math are more likely to select a science-related major [52]. Additionally, studies show that students with higher self-efficacy tend to have greater learning outcomes from their courses [19, 54].

Studies in CS education have also found that self-efficacy is an important factor in determining student persistence and performance in the CS major [15, 56, 57]. For example, Lewis et al. conducted an interview study and found that perception of CS ability is one of the factors that students consider in their decision to major in CS [15]. Historically, studies used traditional self-efficacy surveys to examine the relationship between self-efficacy and persistence [56]. For example, Miura found that students who report higher general self-efficacy on surveys are more likely to take a CS course in college [56, 57]. Studies have also found a significant relationship between programming self-efficacy with both learning outcomes and performance in CS courses [34, 21, 22].

Fortunately, research suggests that a students' self-efficacy can be influenced through both teaching and technological interventions that change one of the four principle sources of self-efficacy discussed above: enactive attainments, modeling of others, social support, and physiological states [23]. For example, increased awareness of mastery of a task can increase self-efficacy as prior studies show that students can better gauge their progress if they are encouraged to reflect on performance mastery and goal accomplishment [23]. Feedback can also influence self-efficacy, as Relich et al. found that students who received attributional feedback on effort and ability showed an increase in self-efficacy and performance compared to other students who followed the same

lessons without the feedback [19]. HCI researchers have found that they can improve student self-efficacy using interventions in both online and in-person environments [58, 59]. For example, online communities offer a unique, public space to showcase progress on task performance [60]. Similarly, studies have shown that sharing work on an online crowdfunding platform can help to build self-efficacy [58, 59].

Interestingly, there are few research studies that attempt to increase student self-efficacy in the CS domain with interventions. There are a number of studies that use self-efficacy measures to evaluate interventions, as self-efficacy is deeply tied to outcomes like engagement [61], persistence [61] and mindset [50, 12, 21]. However, I have not been able to identify any research studies on interventions with the specific goal of improving CS student self-efficacy. Instead, many studies have investigated the internal factors that correlate with self-efficacy. Studies found that self-efficacy correlates with previous programming experiences [62, 22], gender [63, 21, 56], computer literacy [63, 62], metacognitive strategies [21], understanding of programming concepts [64, 22], and sense of belonging in CS [7]. For example, Rammaligan used a survey to measure students' comprehension of software programs and found that students with better mental models of programming concepts were more likely to report a higher programming self-efficacy [22]. Similarly, Askar and Davenport found that students with more years of computer experience had higher programming self-efficacy, and that males on average had higher programming self-efficacy than females [63]. In order to design learning environments and technology that promote positive self-efficacy, then we need to understand how student experiences working on programming problems influence their self-efficacy. There is a gap in the literature as we do not know how students interpret the events that occur in programming with relation to their programming intelligence. Since self-assessments strongly inform self-efficacy, I next discuss existing research on how and when students make self-assessments about their programming ability.

2.3 Self-assessments

In order to realistically inform self-efficacy, self-assessments need to be accurate. Self-assessments are a students' evaluation of their own performance on a task. Self-assessments directly inform a students' enactive attainments, which is the strongest information source to self-efficacy [16]. Generally, when students evaluate their performances as successful, their self-efficacy increases, and when a student views their performances as a failure, their self-efficacy decreases. Accurate self-assessments are also very important for students to be able to self-regulate their learning process [65, 66, 67].

Students do not always have accurate perceptions of their ability. Studies have found that students generally have a self-enhancement bias, evaluating themselves more favorably than others [68, 69, 70]. This was demonstrated when Alicke asked college students to rate the degree to which a set of trait adjectives characterize themselves and characterize the average college student, finding that students rated themselves significantly higher than others for desirable traits [68]. Studies show that this overestimation arises in part due to a lack of metacognitive awareness of one's own weaknesses [71], as well as a natural tendency for humans to be overly optimistic about their abilities [72]. However, not all students have the same positive bias. Self-enhancement biases are often not as strong for students who experience stereotype threat [38], the belief that you are a member of a group that is not fairly represented and stereotypically does not perform well in the domain. For example, Ehrlinger and Dunning gave college students a pop quiz on scientific reasoning and found that female students rated themselves more negatively than male students on scientific skills and estimated performance on the quiz, even though there were no differences in performance based on gender [73].

In researching the student programming experience, studies have found that CS students self-assess their ability frequently [14]. This may be because many CS1 students are new to the field [51] and feel pressure to choose a college major [15]. While evaluating knowledge and monitoring progress are a necessary part of the self-regulated learning process [65, 66, 67], frequent

self-assessing may be problematic to student self-efficacy. One study showed that CS1 students sometimes self-assess negatively even when they are succeeding on a programming problem [14]. Kinnunen and Simon suggest that these negative evaluations may occur when a programming experience does not match the student's expectations. For example, students may believe they have performed poorly if they take longer to solve a problem than expected, even if they complete it successfully. These findings suggest that CS1 students may have unnecessarily high or inaccurate expectations of the programming process, which could negatively influence the frequent self-assessments. A few studies have identified some initial criteria that students may use in these self-assessments, including speed, success of their program, social comparisons, prior experience and grades [15, 14], but we do not know much about the other criteria that students use and how they determine these criteria. We currently do not have a full understanding of how students evaluate their programming ability, particularly throughout the programming session. I address this gap in this dissertation. It is necessary to understand how students evaluate programming intelligence because if students assess themselves using irrelevant or inaccurate metrics, they could have unnecessarily low views of their programming performance and self-efficacy.

2.4 Gap in literature

Each of these theoretical perspectives suggest that students' perceptions of their programming intelligence are influential to their motivation, persistence and experiences in CS, particularly at the introductory level. Researchers that have explored these topics have demonstrated their complexity in their application to the CS education domain. However, we have little insight into how students perceptions of their programming intelligence arises during the programming process itself or how a programming episode influences their self-assessment of programming ability. Thus, we further this research by investigating student experiences in programming episodes through the lens of these three theoretical perspectives.

CHAPTER 3
STUDY 1: HOW DO STUDENTS TALK ABOUT PROGRAMMING INTELLIGENCE?

3.1 Problem and background

Psychology researchers have found that students can have one of two mindsets towards the malleability of intelligence: a fixed mindset or a growth mindset [41, 42, 43]. Students with a fixed mindset believe that intelligence is an inborn trait, and thus value proving their intelligence over learning. These students tend to give up when challenged to avoid failure. In contrast, students with a growth mindset believe that intelligence is malleable, and thus value learning over performance. Students with a growth mindset tend to persist in the face of challenges. In computer science, the fixed mindset is particularly prevalent. Multiple studies have shown that student mindsets become significantly more fixed during their first programming course [12, 11, 10].

Fortunately, interventions that promote the growth mindset have shown to improve student persistence and performance in many domains [44, 46, 47]. These interventions can also reduce the negative impacts of stereotype threat for women and underrepresented minorities [46, 45], which is a known issue in CS1 courses [39]. Unfortunately, we have a limited understanding of how mindsets are enacted in CS, and how to design interventions that improve student perceptions of their intelligence in this context. Multiple studies have attempted traditional mindset interventions in the CS domain with little success [12, 50]. The studies that were able to impact student mindsets were not able to demonstrate improvements in persistence and performance, which were the desired outcomes. In order to design more effective interventions, we need to better understand how CS students think about intelligence, and how these perceptions relate to their persistence and motivation on programming problems.

Within this chapter, we contribute two studies that explore how novice university students define and measure intelligence in CS, and how this affects their perception of programming.

In the first study, we interviewed 9 students about programming intelligence after working on a challenging programming problem. Based on these interviews, we found that only one of our participant's talk aligned with mindset theory; the other eight participants' talk either included both fixed and growth attributes or their talk demonstrated a misalignment between mindset and behaviors. We also noticed that students frequently assessed their programming ability, using surprising criteria like typing speed and ease of debugging. Our second study explored these self-assessment criteria in more depth through a survey of 103 introductory CS students. We found significant variation in the criteria, showing that students define programming intelligence in very different ways. This variation in criteria may explain the limited success of mindset interventions in CS. These findings suggest a need for more research to understand the relationship between self-efficacy and mindsets in CS.

3.2 Interview study

3.2.1 Methods

We designed the first study to explore how novice undergraduate students talk about intelligence in CS. Our goal was to study how this talk reflected students' (1) mindsets, (2) behaviors associated with mindset, like persistence and reaction to challenge, and (3) other motivational factors. We used a qualitative, interview-based approach to address this goal.

Participants and setting

From a mid-sized, private university we recruited nine undergraduate students who were enrolled in CS 1.5, a course designed for students who finished CS1 but did not feel ready to move on to the second course in the CS sequence. We chose this demographic because these students were exposed to programming, but still consider themselves to be novices. Additionally, many students in this course are still deciding whether or not to major in CS, and thus were at a critical junction point in their CS path. Five (55%) of our participants were female, which is representative of the 61% in the course, and two (22%) of our participants had declared CS majors, which is representa-

tive of the 26% of students in the course who had declared. All students provided informed consent to participate and were compensated at a rate of $30 per hour.

Procedure

Our study procedure included three key tasks: a challenging programming problem, the mindset survey, and a clinical interview [74]. First, we asked participants to work on a challenging programming problem for twenty-five minutes. The goal of the programming task was to elicit thoughts and feelings about intelligence that occur only when a student feels challenged, since studies show that mindsets only impact behaviors when students experience a challenge [43]. Additionally, we wanted all participants to encounter a similar experience before the survey and clinical interview so they would be in a comparable emotional state.

Next, we asked participants to complete a mindset survey so that we could compare our qualitative evaluation of their mindset to the canonical mindset measure. We adapted the traditional survey [41] to ask about programming aptitude instead of general intelligence because studies show that domain-specific mindset surveys are more accurate then the original generic intelligence mindset survey [75]. We administered the survey after the programming problem, but before the interview, so that the conversation with the researcher would not impact the survey responses.

Finally, we conducted a clinical interview [74] to elicit a qualitative evaluation of student mindset and associated behaviors in CS indirectly. During the interview, the researcher asked a prepared set of questions and added follow-up questions as relevant topics arose. I started with questions about the programming task (”how well do you think you did on that problem?”) to develop rapport. Then I asked questions about their programming experiences (”tell me about a time when you struggled on a programming problem”) and opinions on programming intelligence (”what do you think it takes to succeed in CS?”) to dive deeply into their viewpoints. At the very end of the interview, I asked about their mindset in CS directly to see if they self-identify with one of the mindsets. I asked about mindset directly at the end of the interview to avoid biasing their responses to the earlier questions.

3.2.2 Data analysis

We analyzed the interview data using a combination of deductive and inductive qualitative coding to create a theoretically informed codebook [76, 77, 78]. We identified the initial deductive codes using research that outlines the relationship between mindsets and behaviors. For example, we created an effort attribution code because studies show that students who have a growth mindset are more likely to attribute their success to effort [51, 42]. We then used open coding to identify emergent themes that we were not expecting based on the mindset literature. From the data, we identified inductive codes for other types of talk that related to mindset, intelligence, and persistence. We iterated on our codes until their definitions were clear. Then, my collaborator and I independently coded all of the data and discussed any discrepancies.

There are 14 codes in the final codebook. Four pairs of codes capture cases where participants' talk exposes their beliefs about the malleability of programming intelligence, which we call mindset talk. For example, when a participant states that there is a limit to their potential growth in CS, we would label this as fixed mindset talk. Three additional pairs of codes capture cases where a participant either behaves or talks about behaving in a way that literature shows is associated with a mindset, which we call associated behaviors. For example, if a participant says that 'struggling while programming is good because it results in learning', we would label it as an associated behavior of the growth mindset. The final codebook is shown in Table 3.1. One limitation in our approach to qualitative coding interviews to study mindsets is that we must interpret students' statements and make judgements about their meaning. While prior research shows that people with certain mindsets tend to say certain things and behave in certain ways, we cannot definitively know an individual's mindset from their talk alone. However, this type of qualitative analysis allows us to gain a deeper understanding of how student beliefs are enacted in real contexts than we can achieve through Likert-scale surveys. We therefore believe this approach will provide important new insights into student mindsets about intelligence in CS.

Growth Mindset Talk Codes	**Fixed Mindset Talk Codes**
Attributes an outcome to effort or learning	Attributes an outcome to their ability
States that growth in CS is possible with effort	States that there is a limit to their ability or potential for success in CS
States that peers are different based on controllable reasons	States that peers are different based on innate reasons
Self-identifies as growth mindset	Self-identifies as fixed mindset
Growth Mindset Associated Behavior Codes	**Fixed Mindset Associated Behavior Codes**
States that struggle, practice, or challenge is good	Does not value effort, struggle, practice, or challenge
Asks researcher for help on programming task	Asks researcher about performance on the programming task
Is motivated, persists, or seeks out a learning opportunity	Avoids a learning opportunity or programming activity

Table 3.1: Qualitative codes used to analyze our interviews, including codes that indicate either a fixed or growth mindset, and codes that indicate behaviors that are associated with either a fixed or growth mindset.

3.2.3 Mindset findings

To analyze how our participants' mindsets are enacted through their talk, we counted the number of statements that were labeled with either mindset talk codes or associated behavior codes for each participant. Then, we calculated the percentage of mindset talk that was coded as growth and the percentage of associated behavior talk that was coded as growth. Based on the literature, we would expect that most students would have one consistent mindset and the corresponding associated behaviors. For example, if the majority of a student's mindset talk was growth, we would also expect most of their associated behavior talk to be growth.

We used the percentages of growth mindset talk and associated behavior talk to classify and cluster the participants. We classified participants as having a growth mindset if at least 75% of their mindset talk was labeled with growth codes, and fixed if 75% of their mindset talk was labeled fixed. We classified participants as having growth mindset associated behaviors if at least 75% of their associated behavior talk was labeled with growth codes, and fixed if 75% of their associated behavior talk was labeled fixed. To confirm that we agreed with the classifications, we read all

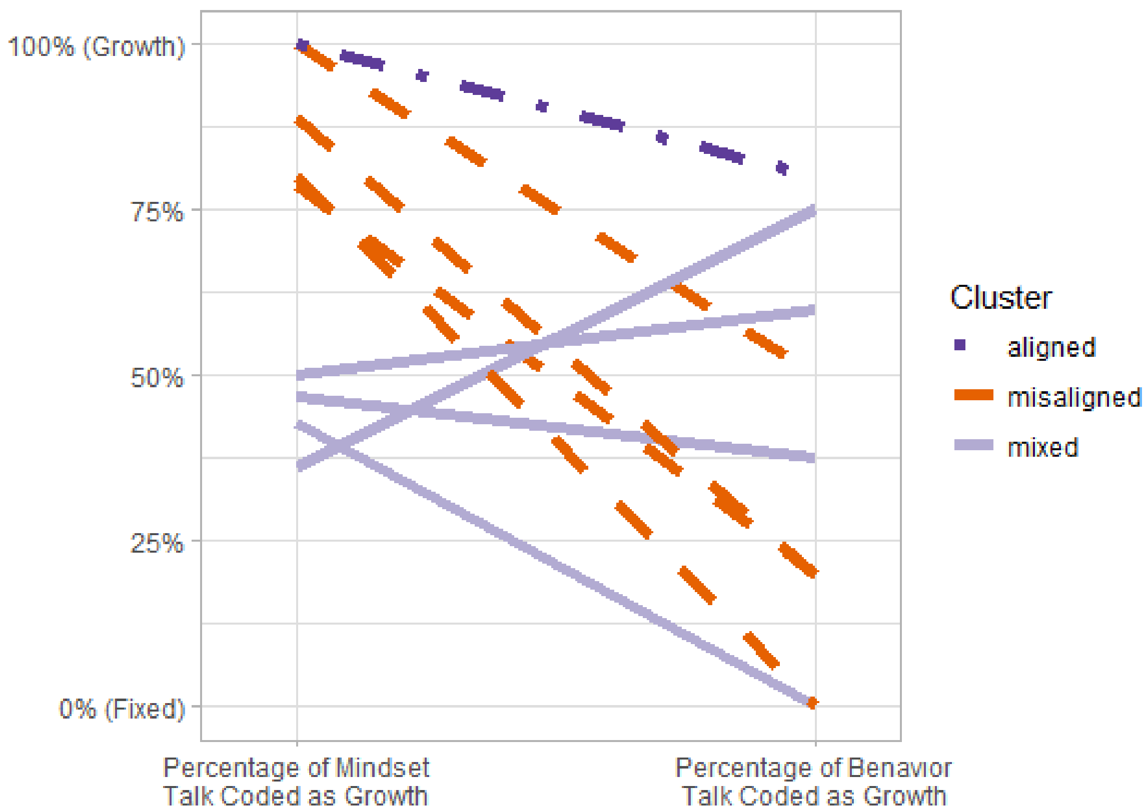


Figure 3.1: Graph showing the percentage of mindset talk that was coded as growth (left) and the percentage of associated behaviors that were coded as growth (right). Participants are grouped into three clusters: the first includes the participant whose talk *aligned* with mindset theory; the second includes participants with *misaligned* mindset talk and associated behaviors; the third includes *mixed* mindset participants.

of the statements coded as mindset talk or associated behaviors for each participant. Finally, we compared our classifications with the participants' self-identified mindsets and responses to the mindset survey.

Next, we identified clusters of participants by analyzing the relationship between their mindset talk and associated behavior classifications, as shown in Figure 3.1. The three clusters that we identified are: *aligned with mindset theory*, *misaligned mindset and associated behaviors*, and *mixed mindset*. We define each cluster in more detail in the sections that follow, and present one participant from each cluster as a case study.

Aligned with mindset theory

This cluster represents participants whose mindset talk is consistent and corresponds with their associated behaviors. We only had one participant in this cluster, P8, a female, first-year student who is not a CS major. P8 talked with a very strong and consistent growth mindset; 100% of her mindset talk was labeled with growth codes. For example, she expressed her belief that effort leads to improvement, saying *"I think I can become better, but I don't think that I am there just yet. . . I put in more hours, so it made me smarter at programming"*. Additionally, all but one of her associated behaviors were labeled with growth codes. For example, she expressed that she seeks out learning opportunities by frequently going to office hours: *"it's very interesting to see how [the TAs] think, because they're so much more experienced, to see how they look through a problem. . . my TAs are really good, they're really cool, they're pretty motivated, and it kinda motivates me to do well"*. Dweck notes that people with a fixed mindset tend to avoid more experienced people because they fear being compared to them and exposed as having lesser ability, while those with a growth mindset actively seek out opportunities to learn from people with more experience [43]. P8's growth mindset talk also aligns with her response to the survey and her self-identified mindset, which were both strongly growth mindset.

Misaligned mindset and associated behaviors

This cluster describes participants whose associated behaviors are misaligned with their mindset talk. Participants in this cluster have a consistent mindset (at least 75% of mindset talk was labeled with one mindset) but their associated behavior talk does not match (less than 75% of associated behaviors were labeled with this mindset). The four participants in this cluster all presented growth mindsets through their talk, but over half of their associated behaviors were fixed.

As a representative example, consider P3, a female, first-year student who is a CS major. 100% of her mindset talk was labeled with growth codes. For example, she saw programming as something that requires learning rather than as an innate skill: *"[Programming is] more about learning the different structure, and the different strategies. I don't think everyone is just born with intuition*

for that. It's a lot of learning." She also believes that if she works hard she can improve: *"If I studied really hard over the summer, I think I would be a lot better. Or at least I'd be more familiar with certain things than my peers."*

However, over half of her associated behaviors were labeled with fixed mindset codes. For example, when asked to share a time when she was proud of something she programmed, P3 talked about a time when she finished an assignment with ease: *"I'm proud of the regular expressions programming assignment, that one wasn't too challenging. There wasn't much programming, but I got it done pretty quickly without that many errors, so I was pretty happy with myself."* Studies show that people with fixed mindsets are more likely to talk about being proud of moments that demonstrate their ability, rather than their effort or learning [79]. In this case, P3 talked about being proud of a moment when she demonstrated her ability instead of one when she learned or overcame a challenge. Some of P3's associated behaviors were also labeled as growth mindset. For example, when describing a programming experience, she saw challenge as an opportunity to grow rather than as a negative reflection of ability when she said: *"I kept getting the error, and I would try and fix it, and then I would get more errors because of fixing that, well, trying to fix that error. And it was just really frustrating, but it was good practice"*. P3 filled out the mindset survey as growth and self-identified as growth. So the mindset survey and the self-report question captured her mindset but did not align with her associated behaviors.

Mixed mindset

The last cluster describes participants whose mindset talk is a mixture of the growth and fixed mindsets; between 25% and 75% of their mindset talk was coded as growth mindset. We categorized four participants as mixed mindset. Their associated behavior talk varied widely, suggesting that the mixed mindset does not correlate with a specific associated behavior profile.

An exemplary case study for this group is P2, a female, third-year student who is not a CS major. She frequently used both growth and fixed mindset talk, making seven and eight statements of each, respectively. She exhibited her mixed beliefs in her response to the question "Can anyone

succeed in CS?", by saying *"I think there are people that are born for this and then there are people that need to try, but then if you try, if you really like it, I think you can. I think there is some advantage to those that their brains are wired in a way"*. In this quote, she demonstrated both growth and fixed mindset ideology as a growth mindset person would believe that if you practice, you can get better and a fixed mindset person would believe that there are people with innate talent in a domain. Similar to her mixed beliefs, her associated behaviors were also mixed. She demonstrated motivation to continue working on the challenging programming task even after the clinical interview, saying *"No. I just want to figure this out"*. On the other hand, we noted five instances of avoidance behavior. For example, she said: *"before I get to other courses that are more fun, I have to go through the theory part of it, and since I'm not dedicated to CS, I don't want to put myself through the unnecessary hard work"*. Studies show that when asked to choose a type of problem to work on, fixed mindset people tend to choose problems that will demonstrate their ability, while growth mindset people pick challenging problems that will foster learning [79]. Unlike her mixed or relatively fixed talk, P2 responded to the survey slightly growth mindset. However, when asked she self-identified as having both mindsets, saying: *"It's like, fixed, in a sense that I think there are people that are meant for it, and then not meant for it. But then, growth at the same time, because even if you're not meant for it, if you try hard enough and if you like it enough, you can always succeed in it. So fixed and growth."*

3.2.4 Self-assessment criteria findings

Beyond classify students' mindsets, we were interested in identifying additional themes from the interview data related to the ways that novice programmers talk about their intelligence. When analyzing the interviews, we noticed that students frequently assessed their programming ability using a wide variety of criteria, which we call *self-assessment criteria*. For example, P9 mentioned the importance of memorizing syntax when he assessed his programming ability, saying *"I feel like I should remember the syntax for basic things, such as lists, and both C++ and Python, more closely than I currently do"*. P8 used the criteria that it is better to do work on your own when she

	Code	Example Quote	Count
Identified in Interview Study	Better if you do it yourself	*"If I go to a...TA and I get a lot of help from them, then I feel kind of bad, because I didn't do the whole program by myself"*	1
	Better if you memorize syntax	*"They know various functions like the back of their hand"*	4
	Faster is better	*"If they can complete an assignment relatively quickly"*	19
	Code quality is important	*"Clean, understandable and short code"*	32
	Computer skills are important	*"By how fast they type"*	7
	Getting errors is bad	*"If it runs the first time they type it out"*	3
	Thinking and planning is not progress	*"If they keep typing and don't have to sit there and think"*	6
Identified in Survey Study	Correct solution	*"[Their] program works"*	5
	Decomposing problems is bad	*"If they can type out a whole long idea and tweak it as opposed to having to do each part piece by piece slowly"*	6
	Decomposing problems is good	*"If they do it in steps, checking/running their code as [they] go"*	7
	Thinking and planning is good	*"They are able to plan out and structure their thoughts on how to approach the code before writing it"*	13
	Good debugging skills	*"They are able to identify bugs...write test cases to check that their code is correct"*	12
	Good articulation skills	*"If they are able to stop and explain to you...what they are doing"*	11
	Ease of debugging	*"They understand how to debug a program quickly based off of first glance"*	15

Table 3.2: Codebook for the self-assessment criteria. The top set of codes were identified during the interview study. The bottom set of codes were identified in the survey study. Both sets of codes were used to code the open-ended survey question. The number in the right column represents the number of times each criterion was identified in the open-ended survey question.

said: *"I'm particularly proud of that [assignment] because I was able to figure out most of that on my own, and I didn't need as much TA help as I had anticipated"*. We identified seven different criteria that the participants used to evaluate programming ability. Table 3.2 describes the seven

criteria codes and provides example quotes.

3.2.5 Discussion

In our interview study, we found that only one of our participant's talk aligned with our expectations based on mindset research. The other eight participants fell into two categories: those whose behavior talk did not align with their mindset talk and those who expressed both growth and fixed mindsets. These findings may explain some of the surprising and unexpected results of prior mindset research in computer science. Specifically, our results show that some students have mixed mindsets and provide new evidence that these students behave in a range of fixed and growth ways. Additionally, we show that the canonical mindset survey cannot capture mixed mindsets, as there is no response that indicates mixed beliefs. Our mixed mindset participants responded to the survey as growth mindset, fixed mindset, and in between. However, we found that when asked to self-identify with a mindset, all of the mixed mindset participants identified as both fixed and growth, suggesting that self-identification could be a more accurate measure of mindset than the survey. We also found that some students' behaviors were misaligned with their mindset talk, which may help to explain why Cutts et al.'s intervention successfully changed students' responses to mindset surveys but did not impact their associated behaviors. If students' behaviors are not always aligned with their mindset, we would not necessarily expect an intervention that successfully changes student mindsets to have an impact on their behaviors.

These findings are surprising because mindset theory is robust, and has been proven in many different domains and contexts. As a result, we suspect that other motivational factors may be interacting with mindset to produce these inconsistencies. We believe that the frequent self-assessments using surprising self-assessment criteria identified in the interviews could be one factor that interacts with student mindsets. While researchers have mentioned the relationship between self-assessments and motivation in CS in previous work [14, 15], our study revealed specific self-assessment criteria that characterize the ways students evaluate programming-specific behaviors, like being able to memorize syntax or fix bugs quickly. These criteria emerged when students

made assessments of their intelligence in the context of a programming experience. Such assessments, which are often called self-efficacy appraisals in the psychology literature, are particularly common when students are new to a field [23], like our participants. Additionally, university students feel extra pressure because they have to choose a major, which may encourage more frequent self-efficacy appraisals as students consider their programming ability in their decision [15, 80].

We hypothesize that these self-efficacy appraisals may be one factor that interacts with student mindsets in CS. If students feel pressure to assess their own ability, they may choose to behave in ways that allow them to make self-assessments, rather than in ways that align with their mindsets about intelligence. Furthermore, these behaviors may depend on the criteria they use to evaluate programming ability. For example, a student who thinks that people who are smart at programming can solve problems on their own may try to assess their own ability by not asking for help or avoiding using resources even if they have a growth mindset. Thus, behaviours driven by these self-assessment criteria could conflict with mindset associated behaviors and produce effects that do not align with mindset theory.

In the interview study, we found that participants used a wide variety of criteria to assess their ability. However, we only interviewed nine students, and therefore do not know whether these findings generalize to a larger population, or whether students disagree about how to define and assess programming ability. We designed a survey study to develop a deeper understanding of the self-assessment criteria, independent of mindset or programming behaviors.

3.3 Survey study

In this second study, we further explored the self-assessment criteria to understand (1) whether the same criteria exist in a larger sample of students, (2) whether other self-assessment criteria arise that we did not find in the interview study and (3) whether there is consistency or variation in the criteria that students use to measure intelligence in CS. Note that we do not aim to study the relationship between self-assessment criteria, mindsets, and programming behaviors in this study. To answer our questions, we designed a survey with three parts: an open-ended question about how

students assess programming intelligence, 36 forced-choice Likert-scale questions about specific self-assessment criteria, and a mindset survey. We collected data from 103 CS1 students through two iterative rounds. In the first iteration, students answered the open-ended question and the mindset survey questions. In the second iteration, students also responded to Likert-scale questions about the specific self-assessment criteria that arose during the first iteration. This iterative design allowed us to explore the prevalence of a wide set of self-assessment criteria and better understand how students describe the criteria.

3.3.1 Participants and setting

We recruited participants from the CS1 course at a mid-sized private university through the course discussion board and department email list. We conducted the study during the final week of the quarter. On the first iteration of the survey, we received 50 responses. On the second iteration of the survey, we received 56 responses, but discarded three who answered incorrectly to a check question, resulting in 53 usable responses. Of the participants we kept in our sample, 44% were female and 25% were CS majors. This closely represents the demographics of the class, which was 40% female and 18% CS majors. Participants who completed the survey were entered into a raffle for one of five $20 gift cards in each iteration.

3.3.2 Open-ended survey question

We used an open-ended survey question to elicit additional self-assessment criteria from students. We asked students to respond to the following question: *"When watching someone program, how do you know if they are good at programming?"* To design this question, we informally tested a few options, including ones that directly asked how students evaluate themselves, but found that participants elaborated on assessment criteria most when asked about a specific instance of another person programming. Since students often compare themselves to peers when making self-assessments, we believe this question effectively elicits the criteria that our participants think are important for determining programming ability. The open-ended structure allows for free response

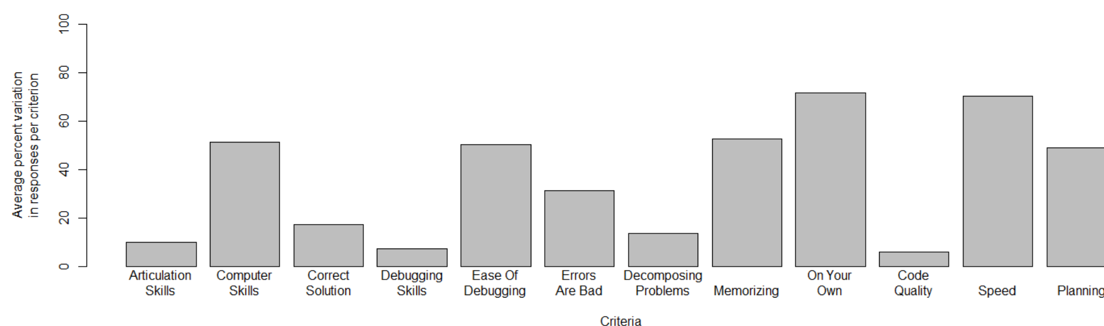


Figure 3.2: Variation in participant responses to self-assessment criteria Likert-scale survey questions, averaged across the three survey questions for each criterion. The variation is calculated by splitting the responses into two groups, agree and disagree, and then computing the inverse of the percent difference of the number of agrees and disagrees: $(1 - abs(agree - disagree)/(agree + disagree))$.

and elaboration, without biasing responses by suggesting particular criteria.

Analysis

We qualitatively coded the responses to the open-ended question for all 103 participants using a combination of inductive and deductive methods [76, 77, 78]. First, we deductively coded the responses using the self-assessment criteria codes identified in the interview study. Then, we inductively coded the responses to identify new emerging themes. My collaborator and I iteratively discussed and refined the codebook, and then each independently coded 10 survey responses (20% of the data). To check inter-rater reliability, we calculated a pooled, prevalence-adjusted kappa, which was 97.5%, signifying excellent agreement [81, 82, 83]. I then coded the remaining data.

Findings

Our analysis of the open-ended survey question revealed instances of participants using all seven of the self-assessment criteria identified in the interview study. Seven new criteria also emerged. The full codebook of criteria can be found in Table 3.2, along with example quotes from students' responses.

Interestingly, two of the new criteria are opposites of ones identified in the interview study, suggesting that participants disagree about these criteria. For example, some participants expressed that decomposing problems is good (*"if they do it in steps, checking/running their code as [they] go"*), while others expressed that decomposing problems is bad (*"they... think about it quickly and write it all in one sequence after thinking"*). However, the converses of the other criteria rarely or never came up. For example, only one participant indicated that using resources is an important part of coding, which could be considered a converse of the code *better if you do it yourself.* These findings suggest that our participants may agree about some self-assessment criteria, but disagree about others. Given the nature of open-ended questions, we can not know if participants agree or disagree with a criterion unless they explicitly mention it, since the absence of a criterion does not necessarily imply disagreement. To test for disagreement in the criteria, we conducted a second iteration of the survey, in which we added Likert-scale questions that directly ask participants about their perspectives on the criteria.

3.3.3 Likert scale survey questions

We designed a set of forced-choice Likert-scale questions to measure whether our participants agreed with statements related to the 14 self-assessment criteria that we identified through the interview study and first iteration of the survey study (see Table 3.2). We designed three questions for each criterion; two that expressed the criterion and one that expressed the converse of the criterion, by building on the quotes and code definitions collected in the first parts of this chapter. The two pairs of criteria that were opposites of each other were expressed through three questions rather than six, resulting in a total of 36 questions. We also included one check item that instructed students to answer 'disagree' to confirm that they carefully read the questions. We conducted think-alouds with students to test if the questions were clear and elicited the desired constructs [84, 28]. An example question for the criterion *faster is better* is: *"If you are faster at solving programming problems, then you are more intelligent at programming"*. An example of a question for *ease of debugging* is: *"Being able to fix a bug easily is an indication of programming intelligence"*. The

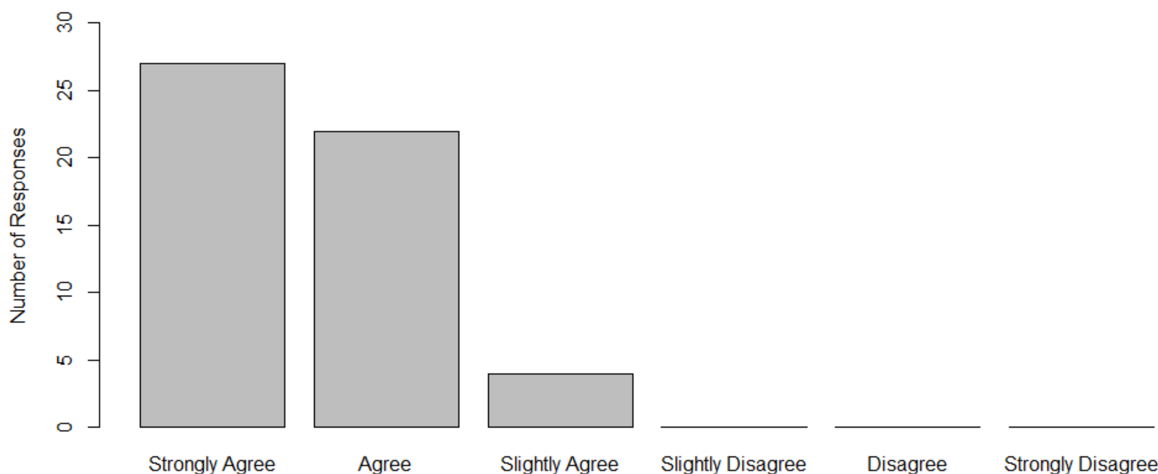


Figure 3.3: Histogram of responses to the Likert-scale survey question: *Being able to explain your program is an indication of programming intelligence.* All participants agreed.

Likert-scale questions were given to the 53 participants in our second round of testing.

Analysis

To clean the Likert-scale data, we flipped the responses to the converse questions, so that all numerical responses represented agreement with the criteria. Then, we analyzed the distribution of responses to each question using bar graphs similar to the one shown in Figure 3.3. To understand the variation in participant responses, we split the responses into two groups, agree and disagree. Then, we calculated the inverse of the percent difference in number of agrees and disagrees to capture the amount of variation in responses. For example, given 20 survey responses in which 10 participants disagree with an item and 10 agree with an item, we would calculate $1 - abs(10 - 10)/20 = 1$, representing the maximum possible variation. However, given 20 participants who agree with an item and 0 who disagree, we would calculate $1 - abs(20 - 0)/20 = 0$, representing the minimum possible variation.

Findings

We found that participants consistently agreed with some self-assessment criteria, but expressed disagreement in response to other criteria. Figure 3.2, shows the average variation in student

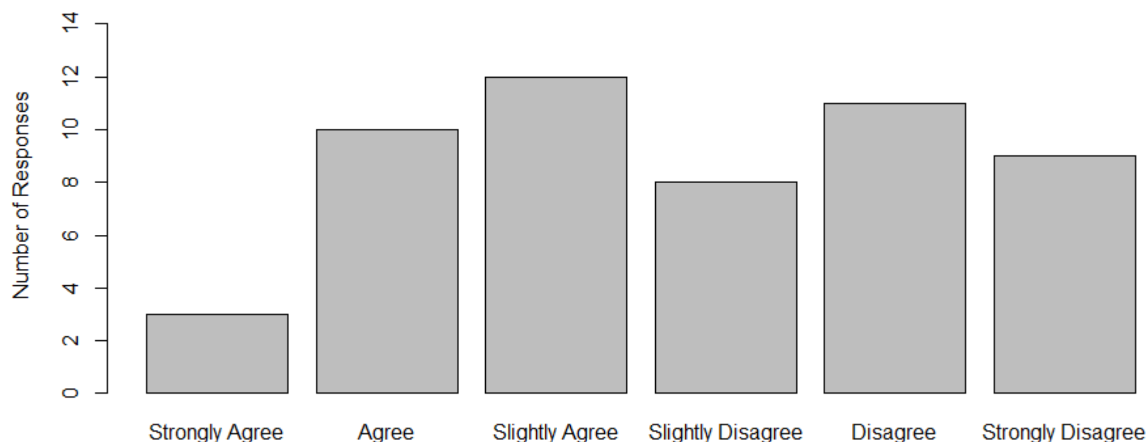


Figure 3.4: Histogram of responses to the Likert-scale survey question: *Someone is more intelligent at programming if they do an assignment on their own, rather than getting help to solve it.* Participants' responses are bimodally distributed, ranging from strongly disagree to strongly agree.

responses to the three forced-choice Likert scale questions for each criterion. Most of our participants agreed with three of the self-assessment criteria; *good articulation skills*, *good debugging skills*, and *code quality is important*. They each had less than 10 percent variation in responses on average, showing that the majority of our participants thought that those skills demonstrate programming ability. Figure 3.3 shows the histogram of responses to one of the *good articulation skills* survey questions, demonstrating that all of the students agreed with the statement. On the other hand, six of the self-assessment criteria had over 50 percent variation in responses, indicating that students have differing perspectives on if those criteria should be used to measure programming intelligence. Figure 3.4 shows the histogram of responses to one of the *better if you do it on your own* survey questions, demonstrating participants' wide range of beliefs.

While this study was not designed to uncover the relationship between self-assessment criteria and mindsets, we were interested in measuring whether any criteria correlated with a particular mindset. However less than 5% of participants reported fixed mindsets on the survey, so we did not have enough data to explore this question. Given the variation in student responses, it is clear that growth mindset students do not all agree with the same self-assessment criteria, so the criteria students use could be one factor that interacts with mindsets to influence programming behaviors.

3.4 Conclusion

In this chapter, we present the results of two studies. In the first study, we interviewed students after they worked on a challenging programming problem and discussed their beliefs about programming intelligence. In the second study, we conducted a survey that asked about the criteria that students use to evaluate programming intelligence. In the interviews, we found that only one participant's talk aligned with mindset theory; the other eight participants' talk either included both fixed and growth attributes or misaligned with their associated behaviors. This is surprising given the robustness of mindset research, indicating there may be an additional factor influencing the enactment of mindsets and persistence behaviors in the domain of CS. During the interviews, we also found that students frequently made self-efficacy appraisals using a variety of criteria. Our findings in the survey study confirm that students define and measure programming intelligence in different ways. We suggest that these criteria may interact with students' mindsets and influence their behaviors. These self-assessment criteria could have a particularly strong impact on university CS students because they frequently make self-efficacy appraisals while deciding whether to pursue a major or career in CS.

While these initial results provide valuable insights about mindsets in CS, there are a number of limitations. First, our interview study had a small number of participants, so we do not know how the mindset clusters that we identified will generalize. Additionally, our analysis depends on qualitatively coding students' talk rather than directly analyzing their behavior during programming, and it is possible that students' talk does not always reflect their behavior in practice. Finally, we recruited participants from the same institution, so we do not know if there are environmental factors influencing the results.

In the next chapter, we will address some of these limitations. First, we expand the scale and demographics of our participants by evaluating self-assessment criteria with students from multiple schools. Additionally, since many of the self-assessment criteria are related to particular moments in the programming process, we analyze student self-assessments relative to particular moments.

Finally, to better understand the influence of these self-assessments on student motivation, we explore the relationship between self-assessments and self-efficacy.

CHAPTER 4
STUDY 2: WHY DO CS1 STUDENTS THINK THEY'RE BAD AT PROGRAMMING? INVESTIGATING SELF-EFFICACY AND SELF-ASSESSMENTS AT THREE UNIVERSITIES

4.1 Problem and background

Foundational research in psychology has shown that students' self-assessments of their ability have a strong impact on self-efficacy [23]. One study found that CS1 students, at times, assess their programming ability negatively even after a successful programming performance [14]. These authors suggest that the negative evaluations occur when students' problem-solving process does not match their expectations [14]. For example, students might think they have performed poorly on a programming problem if it takes longer to complete than they expected.

In the work presented in Chapter 1, we found that students often have varying perspectives on the criteria that should be used to evaluate programming ability. Surprisingly, many of these criteria contradict the practices that instructors think are important to novice success [85, 86, 87], or the practices of professional programmers [88, 89, 90, 87, 91, 92]. For example, we found that some students think looking up syntax and getting errors are signs of low programming ability [13]. These criteria may explain why students have negative reactions to positive programming episodes. Even though many of the criteria are associated with specific moments while programming, we do not know if students negatively self-assess when they encounter those moments and if these self-assessments impact self-efficacy.

In addition, we do not know why students use these self-assessment criteria or how they establish their expectations for the programming process [14]. Since in Chapter 3, we found that students use self-assessment criteria that do not align with professional programming practice [13], we hypothesize that students may have inaccurate perceptions of professional programmers. Ad-

ditionally, prior research indicates that students tend to rate themselves more favorably than their peers, using a self-enhancement bias in their evaluations [68, 70]. At the same time, women tend to under-evaluate their performance in science [73], and have a weaker self-enhancement bias [93]. Given the challenging learning context that CS1 presents for many students, we were interested in understanding if students use self-enhancement biases in their self-assessments and how factors such as gender might shape or encourage the negative self-assessments.

To address these questions, we conducted a survey study with CS1 students at three universities. The goals of this study were to (1) identify the programming moments that cause students to negatively self-assess, (2) investigate differences in how students self-assess across universities, (3) identify whether there is a relationship between negative self-assessments and self-efficacy, (4) explore if students have inaccurate perceptions of professional programming practices and if these perceptions correlate with the moments during the programming process that students negatively self-assess, (5) explore how students assess themselves in comparison to how they assess others and (6) investigate how gender might play a role in students' self-assessments. We designed vignette-style survey questions to measure whether students negatively self-assess at thirteen programming moments, such as struggling with errors or spending time planning.

4.2 Methods

In this chapter, we ask six research questions to better understand the moments that cause students to negatively assess their programming ability:

- **RQ1:** *When presented with scenarios of programming moments, which do students say cause them to negatively self-assess?*
- **RQ2:** *Are there any differences in the moments that students say cause them to negatively self-assess across university contexts?*
- **RQ3:** *Do students who report making negative self-assessments in response to more moments have a lower self-efficacy in their CS course?*

- **RQ4:** *Do student perceptions of professional programming practices correlate with the moments that cause them to negatively self-assess?*
- **RQ5:** *Are there differences in students' assessments of themselves and their assessments of others?*
- **RQ6:** *Are the differences in these assessments impacted by the gender of the student or the character?*

Through these questions, we aim to identify the programming moments that prompt students to negatively self-assess, and uncover any differences across university contexts. We also aim to measure the relationship between self-assessments and student self-efficacy, and determine whether students' inaccurate perceptions of professional programming practice contribute to their self-assessments. Finally, we aim to understand whether any self-assessment biases might explain the prevalence of the negative self-assessments while programming in CS1.

To answer these questions, we conducted a survey study. We chose a survey methodology because it ensures that we can measure each student's reactions to the same set of moments, rather than an observational approach where similar events might not naturally occur for each student. We were interested in learning how responses might vary across different populations of students, so we conducted our study at three universities. Additionally, we conducted follow-up interviews with a small portion of the survey participants to learn more about their thought process when answering the survey questions.

4.2.1 Survey design

We designed the survey with a number of sections, each measuring a different construct. We measured student self-efficacy in their programming course first, to ensure that responses would be unbiased by the later sections of the survey. We adapted the five-question general academic efficacy survey from [94] by changing their references to "coursework" to "in my CS1 class" (where "CS1" is replaced with the name of the student's course). For example, we asked students

Self-assessment moment	Vignette
Getting a simple error	Jen is working on her programming assignment. She runs her code. An error pops up. She immediately realizes that she left out a parenthesis. She adds the parentheses and her code runs successfully. Jen thinks: “That was a stupid mistake. A good programmer wouldn’t make small mistakes like this.”
Starting over	Nadia is working on a hard homework problem. She plans out a solution. She writes a few lines of code. She realizes that her approach to the problem will not work. She decides to start over. Nadia feels frustrated that she wasted time. She erases all her code and starts again.
Not understanding an error message	Frank is working on a programming problem. He runs his code. An error pops up. Frank has no idea what the error message means. He is not sure what to try next. He thinks: “I’m doing so badly, I don’t even know what this message means”.
Stopping programming to plan	Diego starts working on a programming problem. He writes a few lines of code. He realizes that he is confused about what to do next. He pauses and plans his next steps. Diego wishes that he did not have to stop writing code to plan.
Getting help from others	Julie is working on her homework assignment. She gets stuck. Julie meets with an instructor to get help in order to finish the assignment.
Spending a long time on a problem	Tamyra is working really hard on a programming problem. She solves the problem. She is proud of herself. Tamyra looks at the clock and realizes how many hours she spent on the problem. She feels upset because it took her so long to finish it.
Not knowing how to start	Miguel reads his programming homework assignment. He opens up the editor but has no idea where to start. Miguel feels disappointed in himself because he doesn’t even know how to approach the problem.
Using resources to look up syntax	Arjun is working on a programming problem. He can’t remember the syntax. He uses Google to look up the syntax. He is disappointed that he could not remember the syntax on his own.
Spending time planning at the beginning	Jake is unsure how to begin his programming assignment. He spends time planning how to solve the problem. Eventually, Jake comes up with a plan and begins to write code. Jake wishes that he did not need to spend as much time planning before writing code.
Spending a long time looking for a simple error	Isabella is working on a challenging problem. She runs into an error. She looks through the code but can’t find it. After a long time, she realizes that it was a small typo. She thinks to herself: “Wow. I am so bad at programming. A good programmer would not take so long to find a simple error.”
Struggling to fix errors	Daniel is working on his programming homework. He runs his code and gets an error. He struggles to fix the error for a long time. When he runs the code, another error comes up. He struggles again. Eventually, he fixes it. Then, a different error comes up.
Not able to finish in time expected	Sirena is working on her programming assignment. She expects to finish it in one night. After a while, she decides to stop working because it got late. She feels upset that she was not able to finish it in one night.
Does not understand the problem statement	Fatima reads her programming homework assignment. She does not understand what the problem statement is asking her to do. She feels upset and frustrated because she can’t even understand the question.

Table 4.1: The thirteen self-assessment moments and the vignettes that we included on the survey.

to rate how much they agree with the statement, *"I'm certain I can master the skills taught in my CS1 class this term"* on a 6-point forced-choice Likert scale. We chose to use a survey that measured self-efficacy in their CS course instead of a survey about programming self-efficacy in general, because course expectations are well-defined and consistent. In comparison, students may interpret the definition of "good" in a programming self-efficacy survey differently. For example, some students might report a low self-efficacy if they believe they are not good at programming relative to experts, even if they have high self-efficacy about their ability to learn CS and succeed in the course.

Next, we measured whether specific programming moments elicit negative self-assessments. We designed vignette-style survey questions that convey vivid descriptions of specific programming moments and then asked students to report how they feel when they experience those moments. To design the vignette-style survey questions, we curated a list of self-assessment moments. We define a self-assessment moment as a point in the programming process that might elicit student evaluations about how they are doing on a task. We started with the list of moments that are associated with the programming intelligence self-assessment criteria identifed in Chapter 3 [13] and added additional moments based on need-finding interviews with students. Then, we designed vignettes that describe a character encountering each of the programming self-assessment moments. To refine the vignettes and ensure that they represented the moments accurately, we conducted preliminary user studies that asked students to talk out loud while reading the questions to reveal their interpretations. The moments and the associated vignettes can be seen in Table 4.1.

After each vignette, we asked participants to rate how much they agree with two statements: one about the character and one about themselves. The statement about themselves asks students to rate if they negatively evaluate themselves when they experience a similar moment while programming. For example, the statement following the *using resources to look up syntax* vignette is: *I feel like I'm not doing well on a problem when I can't remember the syntax and have to look it up. (6-point Likert scale).* We call these questions *self-assessment vignette questions*. The statement about the character for the same vignette asks students to rate how much they agree with: *I think*

Arjun is not doing well on the assignment because he had to use a resource to look up syntax.. Since we wanted to understand when students were negatively evaluating themselves, potentially causing feelings of low self-efficacy, we only asked about negative reactions to the vignettes. We also included a check question after two of the vignettes that instructed participants to enter a specific response to ensure that they were reading the survey questions carefully.

The gender of the vignette character was communicated through the pronouns used in the vignette. To control for any biases in participant responses based on the character's gender or ethnicity, we randomized the names of the characters across vignettes and participants. We also randomized the order that the vignettes appeared to control for earlier vignettes affecting responses to later vignettes. Finally, we randomized the order that the questions appeared after each vignette.

We chose to use vignette-style survey questions because they help elicit students' memories of similar experiences. This style of survey has been used extensively in the field of psychology as an approach to reduce self-report biases, particularly with survey questions in cases where participants might be concerned about social approval from the researcher [95, 96]. Vignette-style survey questions are also useful when asking about decisions, judgements or situations that the participants may not have previously considered, like when students self-assess while programming. In those cases, asking a direct question might lead to inaccurate results compared to a vignette. However, since vignette-style surveys ask students to recall memories, they can not generate the same emotional experience as a field experiment [97, 98, 29].

The last section of the survey measures whether students believe that professionals encounter the moments described in the self-assessment vignette questions. We hypothesize that students strive to be like professionals in their field and thus their perceptions of professional programmers might correlate with the moments that prompt them to negatively self-assess. The results from our previous study indicated that students do not believe that "good programmers" encounter the self-assessment moments, even though studies show that professional programmers regularly encounter them [88, 92, 87, 13]. We asked about professional programmers rather than more advanced students or "good programmers" because we wanted to capture students' perceptions of undeniable

experts and not of people who are better but may still be learning.

For each of the thirteen self-assessment moments, we asked students to finish a sentence about professional programmers. For example, the question associated with the *using resources to look up syntax* self-assessment moment states:

Professional programmers:

- often forget the exact syntax and use Google and other resources to help them remember.
- remember the syntax they need and rarely have to look it up.

In each question, one option states that professional programmers encounter the self-assessment moment, aligning with research on professional programming practice. The other option states that the self-assessment moment rarely or never occurs in professional practice, which contradicts research on professional programming practice. The questions are presented in a randomized order to control for earlier questions affecting students' responses to later questions. The options following the questions are also presented in a randomized order to control for potential bias caused by the order in which students read them.

The full survey can be found at https://bit.ly/2B7irzC.

4.2.2 Participants

I recruited participants from CS1 courses at three universities that serve different populations of students, all located within the same metropolitan area in the midwestern United States. University 1 is a highly selective, private, research-focused university (R1) that is mid-sized (8,200 undergraduates) and primarily residential. University 2 is a selective, private research university (R2) that is larger (14,500 undergraduates) and primarily nonresidential. University 3 is a less selective, public university with masters programs (M1) that is mid-sized (6,400 undergraduates) and primarily nonresidential. University 3 has been recognized as the most diverse university in the midwest. See Table 4.2 for the demographics of our participants at each university.

School	#	F	AA	A	LA	W	O	2+
University 1	78	58%	6%	35%	4%	42%	4%	8%
University 2	57	32%	7%	23%	10%	47%	5%	5%
University 3	78	17%	7%	24%	31%	24%	8%	6%

 # = Total responses, F = Female, AA = African American, A = Asian, LA = Latin American, W = White, O = Other, 2+ = Two or More Races.

Table 4.2: Demographic data from survey participants for each university in percentages.

A total of 283 students took our survey over a three week period in the middle of their programming courses. I removed data from participants who did not answer the check questions correctly. This left us with a total of 214 participants, with 78 from University 1, 57 from University 2, and 79 from University 3. I recruited participants from two introductory CS courses at University 1, one that is required for all CS majors and another that is not part of the major sequence. I recruited students from the introductory programming course at University 2. I recruited students from the first two classes in the introductory computer science sequence for CS majors at University 3. Since I was most interested in studying self-evaluations across the different school contexts, we focus our analysis at the school level rather than the class level.

4.2.3 Survey procedure

All participants completed the survey in a proctored room with a researcher present. However, the recruitment and distribution of surveys varied based on the constraints of the university policies, course structure, and instructor preferences. For some of the classes, I announced the survey in class and posted the announcement on the class discussion board, directing interested students to take the survey with the researcher at a designated time and place on campus, outside of class time. For other classes, I announced the survey in class and interested students filled out the survey at the end of class. For the remainder of the classes, I announced the survey in the course lab section and interested students filled out the survey during or directly after that lab section. These different recruitment methods may have impacted the participation rates in the classes, so our findings may be subject to participation bias. However, given that this is the first study of self-assessments across

multiple universities, we believe it still makes an important contribution despite this limitation. All survey participants provided informed consent for the survey and were compensated for their time with a $5 gift card of their choice.

4.2.4 Follow-up interview procedure

I conducted semi-structured follow-up interviews to understand whether students interpreted the vignette-style survey questions in the ways we expected. I also explored whether the vignettes encouraged students to recall similar moments in their own programming experiences. Finally, I used the interviews to investigate students' rationales for their responses. I randomly selected survey respondents from each of the three universities to participate in the follow-up interviews from the participants who indicated interest in participating in future research studies. Of the students I contacted, I interviewed 6 participants from University 1, 4 from University 2 and 3 from University 3.

Students who agreed to participate in the interview met with the me individually, either via video conference or in-person at their respective university. They first signed a consent form allowing for audio and video recording of the interview. Then, I gave the participants their previously completed survey and asked them to re-read each of the vignette questions. I asked them to think-aloud while reading the questions and say anything that came to mind, similar to a talk-aloud [84]. I then asked students to explain why they answered each of the survey questions the way they did.

4.2.5 Findings

To answer our four research questions, we analyzed a number of measures calculated from the survey responses. Before beginning the analysis, we evaluated the normality of our data using a Shapiro-Wilk test and found that it was statistically significant for all of our measures. We therefore use non-parametric statistical methods for the analysis presented in this chapter.

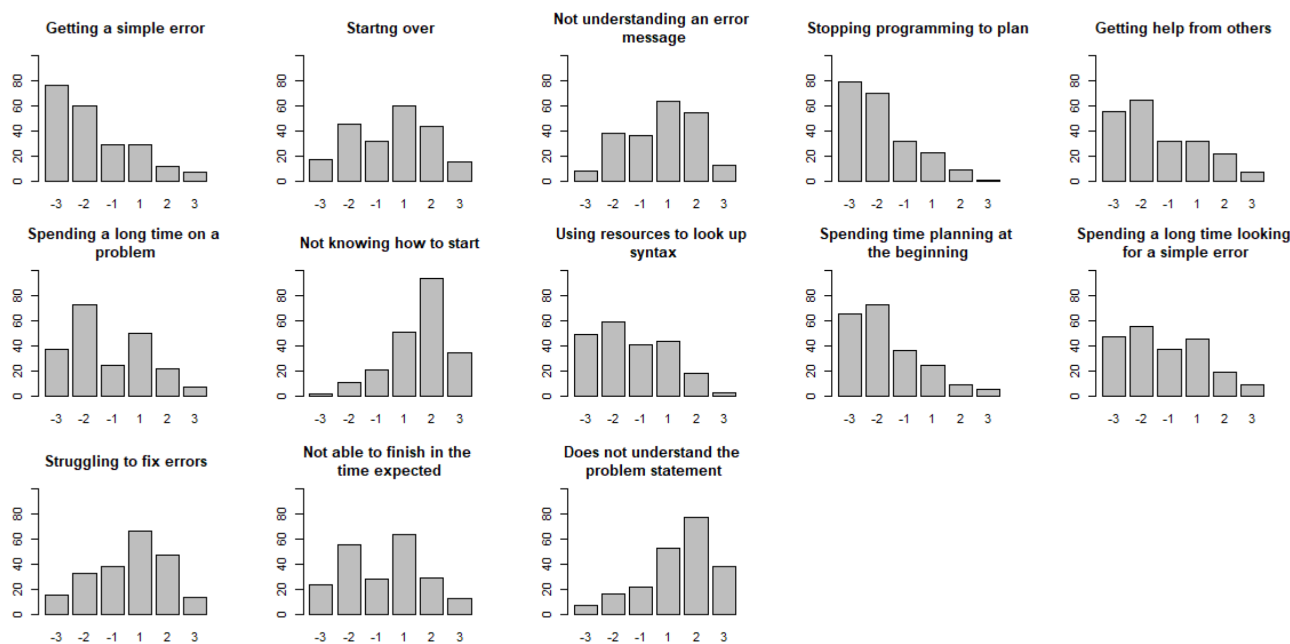


Figure 4.1: Histograms of the responses to each of the self-assessment vignette questions on a 6-point Likert scale ranging from strongly disagree (-3) to strongly agree (3). Students who answered on the agree side of the scale were reporting that they negatively self-assess at that programming moment.

4.2.6 Students from all three universities reported negative self-assessments

To answer RQ1, we analyzed students' responses to the self-assessment vignette questions to evaluate if these moments prompt negative self-assessments. First, we plotted a histogram of student responses to each self-assessment vignette question, shown in Figure 4.1. We expected that only some students would report that they negatively self-assess at each moment because in our previous study, we found high variation in students' agreement with the self-assessment criteria [13]. The histograms show that the responses to some of the self-assessment vignette questions were widely distributed across the scale, like *struggling to fix errors*, while others leaned heavily to one side, like *does not understand the problem statement*. A few of the questions even have a bimodal distribution, like *starting over*, which shows that students report two distinct views of these moments. Overall, we found that each question has at least some responses across the full scale, indicating that all of the moments prompt negative self-assessments for some students. These results are

notable because the moments described in the vignettes are natural parts of expert programming practice, and should not necessarily prompt students to negatively self-assess.

To gain an overarching view of the directionality of student responses, we calculated the percentage of participants who agreed with each self-assessment vignette question by combining participants who responded with *slightly agree*, *agree*, and *strongly agree*. Agreeing with a question represents a negative self-assessment at that moment. A summary of the percent agreement for each self-assessment vignette question is shown in Table 4.3. The results ranged from 15% of students reporting that *stopping programming to plan* prompts negative self-assessments to 84% of students reporting that *not knowing how to start* prompts negative self-assessments. These results help us identify the programming moments that most students use to assess their ability.

To answer RQ2, we analyzed whether there were any differences in the moments that students say cause them to negatively self-assess across university contexts. We enumerated their responses to the 6-point Likert scale, ranging from negative three for *strongly disagree* up to positive three for *strongly agree*. We used a Kruskal-Wallis test to compare students' responses at each university for each of the self-assessment vignette questions, the results of which are shown in Table 4.3. Out of the thirteen self-assessment moments, only one showed a significant difference between the three universities: *finishing in the time expected* ($H(2) = 10.16$, p-value = .006). For this self-assessment vignette question, the mean responses were 0.077 for University 1, 0.053 for University 2 and -0.760 for University 3, suggesting that fewer students at University 3 view completion speed as a sign of ability, compared to the students at Universities 1 and 2. It is surprising that only one of the self-assessment moments was significantly different across the three universities because these are different types of institutions serving different populations of students. This suggests that most of these moments can be generalized across contexts.

4.2.7 Students understood and related to the vignettes

We analyzed the twelve interviews to ensure that students interpreted the vignettes in the ways we intended, and to evaluate potential risks in the survey design. We did not conduct a formal

Self-assessment moment	Percentage that self-assesses at the moment	Comparison of responses to self-assessment vignette questions across universities	Percentage with inaccurate perception of professionals	Relationship between each self-assessment vignette and the associated professional programmers question
Getting a simple error	22.43%	(H(2) = 0.267, p = 0.875)	19.16%	U(214) = 3823, p = 0.421
Starting over	55.61%	(H(2) = 1.328, p = 0.515)	14.49%	U(214) = 3196.5, p = 0.249
Not understanding an error message	61.68%	(H(2) = 0.374, p = 0.829)	57.94%	U(214) = 6025.5, p = 0.306
Stopping programming to plan	15.42%	(H(2) = 3.859, p = 0.145)	10.28%	**U(214) = 2784.5, p = 0.010**
Getting help from others	28.50%	(H(2) = 4.828, p = 0.089)	26.64%	U(214) = 5171.5, p = 0.074
Spending a long time on a problem	36.92%	(H(2) = 1.034, p = 0.596)	14.95%	**U(214) = 3595.5, p = 0.029**
Not knowing how to start	84.11%	(H(2) = 2.495, p = 0.287)	50.93%	U(214) = 6061, p = 0.430
Using resources to look up syntax	30.37%	(H(2) = 1.227, p = 0.542)	42.06%	**U(214) = 6920, p = 0.002**
Spending time planning at the beginning	18.22%	(H(2) = 4.471, p = 0.1069)	15.42%	**U(214) = 4038, p < 0.001**
Spending a long time looking for a simple error	34.58%	(H(2) = 4.051, p = 0.1319)	35.98%	**U(214) = 6246, p = 0.022**
Struggling to fix errors	59.81%	(H(2) = 1.834, p = 0.3997)	44.39%	U(214) = 6445.5, p = 0.071
Not able to finish in time expected	49.53%	**(H(2) = 10.16, p = 0.006)**	17.29%	U(214) = 3760.5, p = 0.1457
Does not understand the problem statement	78.97%	(H(2) = 1.173, p = 0.556)	48.60%	**U(214) = 6688.5, p = 0.027**

Table 4.3: Results of our statistical analysis of the self-assessment moments. The second column displays the percentage of students who agree with each vignette question. The third column shows the Kruskal-Wallis tests evaluating the differences in student responses to the vignette questions across the three schools. All but one of the vignette questions showed no significant difference, suggesting that most of these moments can be generalized across contexts. The fourth column shows the percentage of students who report that professional programmers do not encounter the self-assessment moments. The last column displays the Mann-Whitney U tests evaluating the correlations between responses to the self-assessment vignette questions and associated professional programmers questions. Six of the moments showed significant results, suggesting that these perceptions may contribute to students' self-assessments at these moments. Significant results are bolded.

analysis of the interviews because our sample size was small. Thus, instead of looking for themes, we extracted instances in the interviews when students described their interpretation of a vignette scenario, provided rationale for their response, or mentioned that a vignette sparked the recollection of a particular programming memory. The vignette questions were newly designed and thus we want to confirm the validity of our findings. One potential risk is that students might answer the self-assessment vignette questions based on hypothetical moments rather than memories of specific programming experiences. However, we found that students referenced their own related programming experiences when discussing their answers to the self-assessment vignette questions in the interviews. For example, P6 discussed a specific experience from the current week's homework when explaining his response to the *starting over* vignette:

> *"A lot of the times I have to erase my code, like many-a-times, like even for this current homework assignment I had to erase several lines of code, sometimes just starting from the beginning."*

We also saw participants relate events that happened to the vignette characters to their own programming experiences. For example, P10 noticed that both he and Diego get help from the TAs. He said:

> *"[The question was] kinda cool for me because Diego had to get help from the instructor and I often have to get help from the TAs a lot and I don't feel like I'm doing bad. Like I feel like I'm just improving myself."*

Another potential risk is that students might only answer the questions negatively because the questions are framed with a negative angle. We would expect students to respond in both directions because our findings from Chapter 1 show that they have varied reactions to these moments [13]. To evaluate whether this happened, we looked for instances in the interviews when students explained the rationale for their responses to the questions in both positive and negative directions. We found that students agreed and disagreed with the questions and had a distinct rationale for their

responses. For example, we heard students describe both opinions around the topic of planning. P3 described planning as an important part of the process:

> *"I actually [stop to plan in the middle of a problem] a lot but I don't feel bad about it because I feel that planning is a really huge part before you even start anything. So it is something that you shouldn't feel bad about. I actually think it is one of the things that everybody should do before just jumping into a problem."*

On the other hand, P1 said that when she spends time planning, she feels that it means she is not properly prepared for the problem:

> *"Yeah I definitely feel bad when I have to spend time planning and can't start programming right away, because at that time I am in a situation where I feel that oh I did a lot of practice and still I was stuck and I did not know where to start from and where to end ... as a computer science student, we don't have to think after reading the question, that ok this is the plan going on in my mind. Rather, we should implement it and we should write it whatever way we feel and by running the program, we can get to know the error instead of wasting time in the beginning."*

Both participants personally related to the vignette whether they agreed or disagreed with the character's negative assessment. This suggests that the vignettes elicited reactions and captured the differences in from students who felt both similarly and differently from the character.

Another potential risk is that students might not consider the nuances in the survey question when providing their answer, which could discount the distinctions we incorporated into the vignettes. However, we found that participants mentioned that they weighed these specific details in the vignettes when deciding how to answer the questions. For example, P7 discussed the nuances in the different types of errors:

> *"You are not doing poorly if you are getting a bunch of syntax errors. But sometimes you do get errors that you completely don't understand and don't make any sense and then that is a loss of concept ... I think it would feel more like a setback if it was*

a concept understanding because I feel like that is more part of the understanding process... So, I would feel like I'm doing less well in comparison to where I get a syntax error."

This shows how the nuances in different types of errors impacted the way P7 answered the question, suggesting that students consider these details when reacting to the scenarios. This indicates that incorporating nuanced details into survey questions is important for gaining an accurate representation of the moments that elicit negative self-assessments.

Overall, we found that students resonated with the vignettes and reflected on their own experiences while answering the questions. These interviews helped establish that the self-assessment vignette questions capture students' interpretations of programming moments with reasonable accuracy.

4.2.8 Students who self-assess more frequently have lower self-efficacy

To answer RQ3, we analyzed the relationship between students' responses to the self-assessment vignette questions and the self-efficacy survey. Enactive attainments, or the results of performing tasks related to subject mastery, are the most influential information source for students' self-efficacy [23], so we expected that students who negatively self-assess more strongly or more frequently would have lower self-efficacy in their programming course.

To test this hypothesis, we created a measure that represents the degree to which each student negatively self-assesses, which we call the *self-assessment compound score*. We computed this measure by averaging students' responses to all of the self-assessment vignette questions after converting the responses to numerical values ranging from negative three to positive three. In averaging the questions, we are not suggesting that students' responses should be internally consistent, but rather that they each represent an instance of a negative self-assessment. For example, two students could have the same self-assessment compound score, yet one student might report making negative self-assessments when he gets errors while the other student reports making negative self-assessments when she spends time planning. We therefore use this measure to get an overall

sense of how strongly and frequently each student negatively self-assesses. We also averaged students' responses to the five self-efficacy questions to create a *self-efficacy compound score*, after confirming that the responses have high internal consistency (Cronbach's alpha of 0.91).

Then, to test our hypothesis, we used a Spearman's rank correlation coefficient to measure whether there was a correlation between the two compound scores. The results show a significant negative association between students' self-assessment compound score and their self-efficacy (r_s(212) = -0.418, p $<$ 0.001). This finding shows that students who negatively self-assess more frequently and strongly tend to have lower self-efficacy in their CS1 course. From this analysis, we cannot determine whether this is a causal relationship. However, since self-efficacy theory has previously established that negative self-assessments influence students' overall self-efficacy [23], it is possible that there is a causal relationship between these negative self-assessments and self-efficacy. Future research should therefore try to establish this causal relationship.

We were also interested in measuring whether there were any differences in the relationship between students' negative self-assessments and self-efficacy across the three universities. First, we tested whether there was a difference in the compound self-efficacy scores between the three schools. A Kruskal-Wallis test revealed a significant difference (H(2) = 13.96, p-value $<$ 0.001). Students at University 1 reported an average self-efficacy compound score of 1.808, compared to an average of 1.375 at University 2 and an average of 2.071 at University 3 (higher scores indicate higher self-efficacy). Given there was a difference in students' self-efficacy across the universities, but not in their responses to the self-assessment vignette questions, we hypothesized that there might be a difference in the correlation between the self-assessment compound score and self-efficacy. To test this, we ran a non-parametric ANCOVA (using the sm.ancova subroutine in R), which analyzes differences in correlations between a set of non-parametric regression curves [99]. We chose a smoothing parameter of h=1 because this provided an accurate representation of the data without overfitting. The test returned a p-value $<$ .001, showing that there is a significant difference in the strength of the correlation between the self-assessment compound score and self-efficacy across the three schools. The fit lines in Figure 4.2 show that the self-assessment

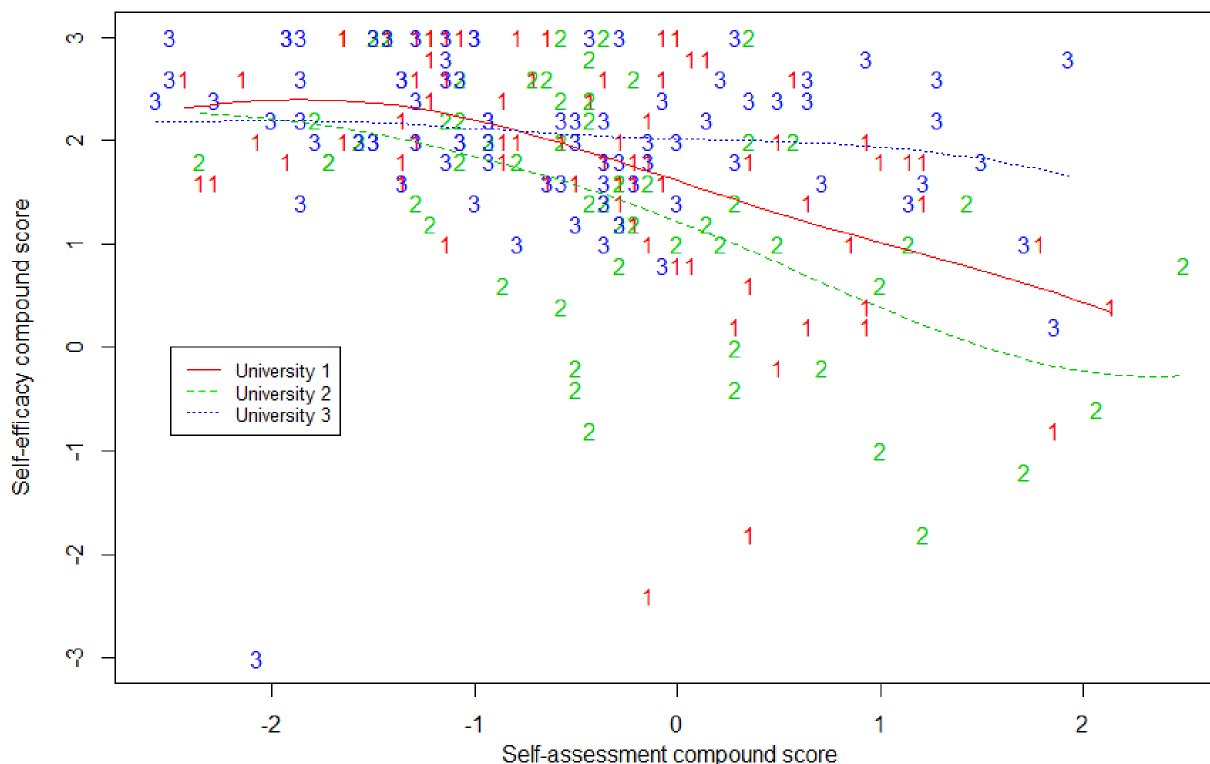


Figure 4.2: Graph showing the self-assessment compound score and the self-efficacy compound score for each survey participant, grouped by university. The results of the nonparametric ANCOVA show that the fit lines for each of the universities are significantly different from each other.

moments have a stronger correlation with self-efficacy for students at University 2, and a weaker correlation for students at University 3. This could be because a number of other factors beyond self-assessments correlate with self-efficacy in computer science [63, 21, 22, 64, 7]. For example, if students already have high self-efficacy due to factors like encouragement from the instructor, high grades, or an expectation that the course is easy, self-assessments may not have as strong of an impact on students' beliefs in their ability to succeed.

4.2.9 Perceptions of professional programmers may influence self-assessment moments

We expect that multiple factors may impact the set of moments students use to negatively self-assess while programming, and findings from our previous study suggested that students' perceptions of more experienced programmers may be one of those factors [13]. Therefore, to answer RQ4, we explored students' perceptions of professional programming practice to evaluate whether

these perceptions correlate with the moments that cause them to negatively self-assess. First, we evaluated how frequently students reported that each of the self-assessment moments rarely occur in professional practice. The results ranged from 10% of students believing that professionals do not stop programming to plan to 58% of students believing that professionals always understand their error messages, shown in Table 4.3. This indicates that there is a wide variation in students' perceptions of professional programmers and shows that many students' perceptions do not align with studies of professional practices [88, 92, 87].

Next, we wanted to measure whether students' responses to the self-assessment vignette questions correlated with their responses to the professional programming survey questions. We hypothesized that students who think that professional programmers do not encounter the self-assessment moments would be more likely to negatively self-assess during these moments. For example, if a student believes that professional programmers do not spend time planning, she may be more likely to negatively self-assess when she has to spend time planning. We used a Mann-Whitney U test to evaluate if there was a significant difference in students' responses to the self-assessment vignette questions based on their responses to the corresponding professional programming questions, as shown in Table 4.3. We found a significant effect for six of the moments, for example *using resources to look up syntax* and *spending time planning in the beginning*. Additionally, two of the other moments were trending towards significance (p-value between .05 and .1). We found no significant effect for five of the moments. These results suggest that in some cases, students' perceptions of the professional programming process may influence the moments when they negatively self-assess.

Finally, we analyzed the interviews to understand whether students' perceptions of professional practice factored into their responses to the self-assessment vignette questions. We also analyzed the interviews with a more exploratory lens to identify other rationales that students provided for their responses. We noticed that multiple students, unprompted, brought up professional programming practice when explaining their response to the self-assessment vignette questions. For example, when P10 explained why she does not negatively self-assess after getting a syntax error,

she said:

> *"This is with simple errors . . . I strongly disagree that you are doing badly on it. I don't even feel like I'm doing badly on it because I get a small error. Again, I'm pretty sure professionals make mistakes."*

Similarly, P9 explained her response about using resources saying:

> *"I said slightly agree because I've gone to a lot of tutoring centers so I know that professionals or people with more experience than me do use websites to help them figure it out. So I feel like I am not doing well [when I need to use resources] because I wish that I could figure it out on my own and I would be doing better if I could figure it out on my own but it's not a major way to see that I'm not doing well."*

Because P9 knows that professional programmers use resources, she responded with *slightly agree* instead of *agree*. While her perception of professional programmers may not have fully informed her reactions to this moment, she still factored it into her response. Additionally, when P8 explained her response to the vignette about memorizing syntax, she said:

> *"I [think] that programmers need to know every single little piece of syntax and every code and how to at least start doing everything. . . so the fact that I don't have it memorized just made me feel bad."*

Note that we cannot tell from this statement whether P8 is referring to professionals or simply more experienced programmers. In either case, it is clear that she is considering her perception of expert practice in making her self-assessment. Overall, these quotes show that students use their perceptions of more experienced programmers to form their assessments of particular programming moments.

Through our analysis, we identified additional factors beyond perceptions of professionals that students considered when reporting on the self-assessment moments. For example, many participants used social comparisons. P13 said:

> *"Yeah I definitely feel badly on a problem if I don't know where to start. Just like I said before, just seeing everyone else around me being able to solve it. I don't know if I'm just struggling and everyone else is breezing through it, then I feel badly about myself and I feel like I'm not as smart as everyone else."*

Self-efficacy theory states that the vicarious experience of watching peers informs students' enactive attainments [23], which could explain why social comparisons arose as a rationale for students' responses. Students also rationalized their responses by citing recommendations given by their professors. For example P4 said:

> *"I put disagree, mainly because of our professors. They always tell us that planning should be first and once you have your plan then you start coding."*

These quotes suggest that peer comparisons and recommendations from professors are additional factors that may contribute to the particular moments that prompt students to negatively self-assess, which should be explored further in future studies. These other factors may help explain why we did not find a correlation between students' responses to the perceptions of professional programmer questions and the self-assessment vignette questions for every moment. Students' inaccurate perceptions of professional programming practice partially explain why students make negative self-assessments at natural parts of the programming process, but other factors also play an important role in determining students' self-assessments.

4.2.10 Students evaluate themselves more critically than they evaluate others

To answer our RQ5, we measured whether there were differences in students' assessments of themselves and their assessments of the vignette characters. We first converted the responses to the two forced-choice Likert-scale questions following each vignette to a numerical scale ranging from -3 (strongly disagree) to 3 (strongly agree). By agreeing to a statement that follows a vignette, participants demonstrate a belief that they (or the characters) are performing poorly during that moment. Therefore, to calculate self-critical bias we subtracted their response to the question

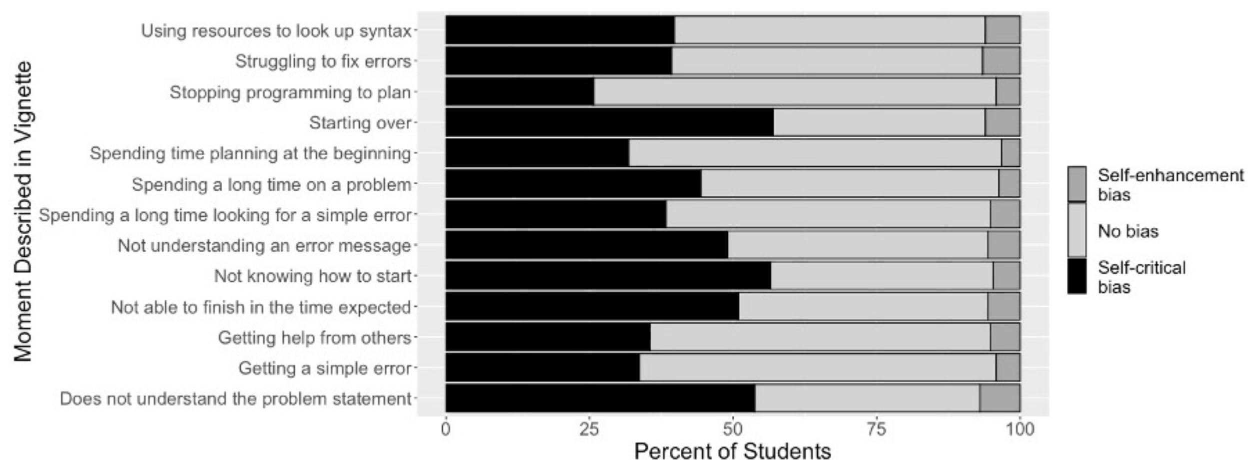


Figure 4.3: Graph showing the percentage of participants who exhibited a self-enhancement bias, a self-critical bias, and no bias, when comparing their responses to the questions following each of the thirteen vignettes.

about themselves from their response to the question about the character for each vignette question. For example, a participant may slightly agree (1) that the character is performing poorly in a particular moment, and slightly disagree (-1) that they are performing poorly. We would calculate 1 minus -1 resulting in a self-enhancement bias of 2 for that vignette. A positive value indicates a self-enhancement bias in participants' responses. After calculating these self-enhancement biases, we grouped participants into three categories for each vignette: those who exhibited a positive self-enhancement bias, those who exhibited no bias, and those who exhibited a negative self-enhancement bias.

Surprisingly, we found that very few students exhibited a self-enhancement bias. As shown in Figure 4.3, only 3-5% of participants had a positive self-enhancement bias for each question, while 25-60% of students had a negative self-enhancement bias. We refer to a negative self-enhancement bias as a self-critical bias. Overall, these findings suggest that CS1 students tend to be more critical of themselves than of others, which is surprising given previous findings on the prevalence of self-enhancement biases. This self-critical bias could manifest in two degrees of severity. Students could believe that both they and the character are performing poorly or well, but differ in the strength of the assessments (e.g. slightly agree for the character, agree for themselves). Or, students could believe that they are performing poorly (e.g. slightly agree) but make no negative assessment

of the character (e.g. disagree), and vice versa. We were interested in measuring how often students assessed themselves in a different direction than the character, so for each vignette, we counted how often participants' responses to the two vignette questions fell on different sides of the Likert scale. We found that only 1-4% of participants negatively assessed the character but not themselves for each vignette, while 9-36% of participants negatively assessed themselves but not the character for each vignette. Furthermore, we found that 85% of students negatively assessed themselves but not the character for at least one vignette, while only 18% of students assessed the character but not themselves for at least one vignette. This effect was most common for the moments *starting over* and *does not understand the problem statement*, in which 35% and 33% of students respectively assessed themselves but not the character negatively. These findings show that a significant number of students negatively assess themselves at moments that they think are acceptable for others.

4.2.11 Self-critical bias is stronger when the student or the vignette character is female

Finally, to answer RQ6, we analyzed whether self-critical bias was influenced by gender. For the remaining analyses, we did not include the five students who reported non-binary gender identities because we feared the size of the group would result in an inaccurate representation of their experience. For the binary students, we conducted a Mann-Whitney U test, and found that female students were significantly more likely to have a self-critical bias than male students ($Z = 3484.5$, $p < 0.001$), with a median bias of 0.46 for male students and 0.92 for female students.

Women are underrepresented in computer science, and prevalent stereotypes depict computer scientists as technologically-oriented males [38]. As a result, we wondered whether students might assess themselves differently in relation to female and male vignette characters. We averaged the self-critical bias that each participant reported for the vignettes with female characters and the vignettes with male characters. Then, we conducted a Wilcoxon signed-rank test, a non-parametric paired t-test, between these two scores and found that participants were significantly more self-critical when the vignette character was female ($Z = 8688$, $p < 0.05$). The median self-critical bias was 0.57 when the vignette character was male, and 0.71 when they were female.

After finding that students are generally more critical of themselves in comparison to female vignette characters, we wondered whether this effect is influenced by the gender of the participant. To answer this question, we conducted an Aligned Rank Transform, a non-parametric ANOVA. We used the same self-critical bias scores for female and male vignette characters described above. We found that the gender of the character ($F(1, 206) = 4.39$, $p < 0.05$) and the gender of the participant ($F(1,206) = 14.90$, $p < 0.001$) both had significant effects on self-critical bias. While this effect appears to be stronger for male students, we did not find a significant interaction ($F(1, 206) = 0.36$, n.s.). We often think of female students as being most affected by stereotypes about who belongs in computer science, however these findings show that male students are also influenced by these narratives.

4.3 Conclusion

In this chapter, we contribute the results of a survey study with 214 CS1 students from three universities. We found for each of the programming moments that some students report to negatively self-assess when they encounter the moment, even though these moments occur in professional practice. Interestingly, for twelve of the thirteen self-assessment vignette questions, there was no significant difference in students' responses based on their university, despite large differences in the populations of students that these universities serve. This suggests that many self-assessment moments generalize across different university populations. We also found that the frequency with which students negatively self-assess correlates with their overall self-efficacy in their programming course. While there was little difference in the self-assessment moments across the schools, the degree to which the self-assessment moments correlated with students' overall self-efficacy significantly differed between the universities, which suggests that environment may have a strong influence on self-efficacy. We conducted two secondary analyses to explore potential explanations for why students may negatively self-assess based on these moments. We found that students have inaccurate perceptions of professional programmers and these perceptions correlated with their responses to some of the self-assessment vignette questions. This suggests that these perceptions

may influence when students negatively self-assess. Additionally, we found that very few students exhibit a self-enhancement bias in this domain. Instead, we found that many students exhibit what we call a self-critical bias, with 96% of participants rating themselves more harshly than the vignette character in response to at least one vignette. We also found that female students are significantly more likely to have a self-critical bias than male students and all students tend to be more self-critical in comparison to a female student versus a male student.

Based on our findings, students may not have an accurate understanding of professional practice, and thus may have misguided expectations of their own programming practice. This gap in knowledge may exist because CS1 courses typically do not teach the cognitive aspects of programming, including problem-solving strategies and programming practices [10]. By explicitly teaching about programming practices in CS1, we may be able to help students build accurate expectations of the programming process and reduce negative self-assessments. Previous studies have taught about programming practices through techniques like direct instruction [10] and by showing recordings of professional programmers [100]. However, we also know that some students negatively self-assess in response to moments despite knowing they are not universal signs of poor performance. These students view the moments as more problematic for themselves than for others. To better support this group of students, CS1 teaching staff could call attention to moments when students may be assessing themselves particularly harshly, and help these students reframe their perceptions of the moments. These interventions will likely improve student self-efficacy in their programming course, since we found that students who negatively self-assess more frequently and strongly tend to have lower self-efficacy.

While these results provide valuable insight into CS1 student experiences, our study has a few limitations. First, even though we chose the self-assessment moments included in our survey based on previous research and preliminary user studies, it is likely that our set of moments is not comprehensive. There may be other moments in the programming process that prompt students to make negative self-assessments. In particular, cultural differences both within and outside of the United States may strongly influence the moments that prompt students to negatively self-

assess. Additionally, while our interviews with a small sample of students provide promising initial evidence that our survey accurately captures student self-assessments, we need to conduct a more formal validation of the survey. Finally, our results rely on student self-reports based on remembered experiences triggered by the vignettes. While retrospective assessments are still relevant for understanding students' perceptions of ability, we do not know whether these responses accurately reflect the thoughts that arise during programming episodes. However, we chose this methodology because the survey allowed us to collect a larger sample of data with consistent experiences between students.

This research lays a theoretical foundation for designing interventions that reduce unnecessary negative self-assessments for novice programmers. With the recent emphasis on personalized feedback interventions through the IDE, a system that automatically detects self-assessment moments from interaction log data could help us study these moments and intervene when they occur. Thus, in the next chapter, I focus on identifying and detecting when these moments will occur, with the goal of providing direction for researchers or practitioners to design interventions that can help reframe student reactions when they occur. Additionally, from this study, we provide both practitioners and researchers a new understanding of the student experience for course instructors and researchers interested in the student programming process.

CHAPTER 5
STUDY 3: AN APPROACH FOR DETECTING STUDENT PERCEPTIONS OF THE PROGRAMMING EXPERIENCE FROM INTERACTION LOG DATA

5.1 Problem and background

In the previous chapters, we established the prevalence of negative self-assessments in CS [31, 13]. However, we do not know how the negative self-assessments arise in and impact the student programming experience. Currently, we have a limited understanding of the programming moments that prompt these negative self-assessments and can not identify them in the programming process as they were defined broadly in the vignette survey questions. If we could detect these moments as they arise during the programming process, we would be able to study the programming experience at those moments more directly.

Furthermore, if such a detection system were automated, we could study these moments using a significantly larger sample of data, as manually detecting them is labor-intensive. An automated detection system would also enable the development of real-time feedback interventions, which provide messages to students at key moments. This type of intervention has been shown to be particularly effective in influencing student beliefs that arise during an activity and mediating students' perceptions in other contexts [48, 49]. Additionally, since the feedback would be provided by a technological system, these interventions can scale to meet the increasing demand and high faculty to student ratios in CS.

Interaction log data collected from programming environments may be useful for automatically detecting self-assessment moments. Researchers have successfully leveraged this type of data to analyze student programming process [101, 102, 103], predict student performance [104, 101, 105, 106, 107], and build automated feedback interventions [108, 61]. However, most of these prior systems use bottom-up methods to identify behavioral patterns in interaction data, rather

than using top-down approaches to detect pre-defined programming moments like struggling with syntax errors, an example of the self-assessment moments. Systems that use top-down approaches, such as cognitive tutors [109, 110, 111] generally require models of expert practice. However, researchers are not experts in understanding how students perceive the programming process, and thus we would need to elicit this knowledge from students to create such a model in this domain.

To address this gap in our ability to study the negative self-assessment moments in the student programming experience, we contribute an approach called *retrospective-enabled perception recognition* for designing systems that detect student perceptions of the programming process. In this approach, the designer uses retrospective interviews [84] to elicit student perceptions of programming moments, and then builds a qualitative codebook that describes the behavioral patterns indicative of each moment. This codebook is used to inform the design of an expert system. We used our approach to design an automated detection system for eight self-assessment moments based on retrospective interviews with 41 CS students. We evaluated the performance of our system using data collected from an additional 33 students, comparing the automatically detected moments to those manually labelled by the authors. Our results are promising, with F1 scores ranging from 66% to 98%. All of the students were from a CS2 class in an R1 university and worked on the same programming problem. Thus, while the results are promising, we recognize that this study was conducted within a very particular context and population, so the system may not be generalizable. We also present an analysis of our systems' incorrect decisions, enabled by the transparency of the expert system approach. Our detection system has the potential to facilitate future studies of self-assessment moments and support interventions that provide real-time feedback. Additionally, we believe the *retrospective-enabled perception recognition* approach can be used more broadly to design detection systems for student perceptions in other contexts.

5.2 Use of interaction log data in CS Education

There is a rich history of research studies that interpret the log data collected from student interactions with programming environments to study the programming process. Researchers analyze

interaction data using two primary approaches: to produce new knowledge from a bottom-up analysis of student interactions (data-driven approaches), and to perform top-down detection of programming moments. I briefly will discuss both of these approaches.

Many researchers have taken data-driven approaches to study the student programming process [102, 103, 112, 113] and to evaluate or predict student performance [104, 101, 105, 106, 107, 20]. For instance, early work by Soloway et al. studied the origins of issues in student programs by logging their compiler errors during programming sessions and analyzing patterns [112]. Similarly, Jadud analyzed compilation logs to measure student progress on programming problems [104]. More recently, researchers have leveraged machine learning techniques to identify patterns in log data. For example, Blikstein et al. clustered students based on their problem-solving pathways to study their progress through programming assignments [101]. Berland et al. also used clustering techniques to study student tinkering behaviors and observe how they change across programming stages [103]. These studies all use data-driven approaches to find patterns in interaction log data that inform our understanding of novice programmers. However, data-driven approaches cannot be used to identify specific moments in the programming process that are predefined based on theory as data-driven decision making processes are not generally human-interpretable.

Fewer studies have analyzed interaction log data with a top-down approach. A top-down approach is where researchers use expert knowledge of the programming process to detect moments from log data. Thus, instead of using an exploratory approach with the data, they use the data to help inform, support, or identify prior theory or empirical findings. Expert systems, which model expert decision-making processes, are a common technique used for top-down approaches. Expert systems reason about student interactions based on models of expert practice and knowledge. For example, expert systems have been used in cognitive tutors, [109, 110, 111] like the LISP tutor [114], to interpret student interaction data and provide relevant feedback. Marwan et al. used this approach to develop a tool that analyze CS student program states to identify milestones in their progress as they solve problems [61]. However, none of these models focus on designing systems that identify moments based on student perceptions instead of expert knowledge. With

our retrospective-enabled perception recognition approach, we suggest a new methodology for building tools that interpret interaction log data based on student perceptions.

5.3 Retrospective-enabled perception recognition

The main contribution of this chapter is our approach for detecting student perceptions of the programming experience from interaction log data. In this section, we describe our new approach and present the methods we used to build a system to detect moments when students may negatively self-assess while programming.

5.3.1 Data collection tools

To enable our system, we designed extensions to collect interaction log data from two programs: jGRASP [115] (an IDE often used in introductory Java courses) and Chrome (a commonly used web browser). We chose these two programs because they account for a large portion of student interactions with the computer while programming. Each extension collects time-series data in a JSON format for a number of user actions and events, which allows us to keep track of student behavior and the state of the IDE. Our jGRASP extension, built in collaboration with the jGRASP development team, captures all keystrokes, cursor movements, console messages, and interactions with buttons and windows. Our Chrome tool captures all navigation on websites, including the URLs and scrolling behavior while viewing a page. During the data collection process, we iterated on the events and actions collected by the extensions as we learned more about the behaviors associated with each moment. For example, after looking at the data, we realized that student scrolling patterns revealed important information about their behavior, so we added this to our extensions.

5.3.2 Phase 1: Retrospective interviews

We conducted retrospective interviews during Phase 1 to capture student perceptions of the programming experience. We recruited 41 participants from a large public university in the United

Table 5.1: Negative self-assessment moments detected by the expert system.

Moments *and detailed descriptions*
Using resources to look up syntax *from the web or other sources*
Using resources to research an approach *from the web or other sources*
Changing approaches *to try a new approach for solving the programming problem*
Writing a plan *in the comments or notes to outline future programming steps*
Getting simple errors *are usually compiler errors due to oversights or typos*
Getting Java errors *are usually runtime errors due to conceptual mistakes*
Struggling with errors *while trying to fix or debug the errors*
Stopping to think *while implementing a solution*

States. At the time of the study, all participants enrolled in a second-semester introductory CS course (CS2), a requirement for CS majors, were eligible to participate. We recruited students with emails sent by the professor of the course. The study took place virtually through Zoom. Students provided consent to participate and were compensated for their time.

The goal of the interview was to gather examples of self-assessment moments naturally occurring during programming sessions, along with participants' perceptions of those moments. When a participant joined the Zoom call, the researcher installed the Chrome and jGRASP extensions on the student's computer. Then the researcher provided a short review of how to use jGRASP to ensure a baseline level of familiarity with the development environment. We asked the student to work on one of three similar programming problems while sharing their screen, and told them to work on the problem like they would a homework assignment. All students programmed using the Java language. During this part of the interview, the researcher turned the student's video and microphone off and did not interrupt them to reduce the effect of the lab environment on their behavior as much as possible.

After 30 minutes of programming, we conducted a retrospective interview [84]. We gave the student a list describing a subset of the self-assessment moments from Chapter 2 [31] (see examples in Table 5.1). We chose to only include the moments that occur during the programming process, like *changing approaches*, and not general reflections, like *spending a long time on a problem*, because we were more likely to be able to determine when they will happen. Finally, the student and researcher watched a screen recording of the programming session and the student identified

Figure 5.1: Timeline graph demonstrating the self-assessment moments that occurred in a participant interview.

each time one of those moments occurred. In Figure 5.1, we provide an example of the self-assessment moments that were labelled in the retrospective interview for one participant.

5.3.3 Phase 2: Qualitative analysis

The goal of Phase 2 was to develop a qualitative codebook that the researchers could use to identify negative self-assessment moments independently, without additional knowledge of student perceptions. Identifying moments such as using resources may appear straightforward, however students' perceptions of these moments are quite nuanced. For example, in our prior work students reported different reactions when using resources to look up syntax versus using resources to research how to solve the problem [13, 31]. While it is relatively easy to determine when a student is viewing a website or a course resource, determining the purpose of its use is more difficult. In addition, it is critical to identify each use of a resource, because a student who references the same resource multiple times will have a different experience than a student who uses multiple resources for different purposes. We therefore use a detailed qualitative codebook to capture the nuances discovered through the retrospective interview process.

To develop this codebook, we qualitatively analyzed the retrospective interviews. After conducting the first 20 interviews, we compiled a list of all student-labeled moments. From that list, we distilled a set of representative behaviors for each moment and wrote an initial draft of the codebook. The codebook includes a high-level definition of each moment and a set of heuristics that describe the behavioral patterns indicative of each moment. We then re-watched the first twenty interviews and iterated on the behavioral descriptions for each moment until two researchers could accurately and consistently label all of the moments. The final codebook can be seen in Appendix A.

As an example, we describe how we identify *struggling with errors* using our codebook. We

defined three levels of behavioral indicators for this moment: strong, medium, and weak. If a student exhibits a strong indicator, such as running code in an attempt to fix a bug three times in a row without succeeding, we would label this as *struggling with errors*. If there is no strong indicator, but there are two medium indicators, such as using resources after getting an error, we would also label this as *struggling with errors*. Finally, while weak indicators, such as a slower pace of typing, are not enough to label the moment on their own, the researchers use them to strengthen their confidence in the decisions.

5.3.4 Phase 3: Codebook verification

In Phase 3, we first tested the codebook using data from an additional 21 interviews. After each new interview, two authors watched the screen recording of the programming session and used the codebook to label the self-assessment moments. Then, the researchers compared their decisions to the participant's labels in the retrospective interview as member-checks of the labelling scheme [116]. When there were misalignments between a participant's labels and the researchers' labels that could not be explained by the participant misusing or missing a label, the researchers adjusted the description of that moment to incorporate the newly observed behavior. This iterative process continued until the researchers did not need to make changes for five consecutive interviews in which the moment was present. At that point, we considered the codebook for that moment to have reached saturation [117, 118]. Of the 12 moments that we asked students to label during the retrospective interview, we were able to reach saturation for eight (see Table 5.1). Most of the moments for which we did not reach saturation occurred at the beginning of the programming session, such as *writing a plan before implementation*. At this point, students generally interact less with the computer, making it more difficult to identify these moments.

5.3.5 Phase 4: Implementation of the detection system

In Phase 4, we built an expert system to detect self-assessment moments using the heuristics in our qualitative codebook. Our system has two stages: data transformation and decision-making. In this

section, I first describe the implementation of each of these stages. Then, I describe an example for detecting one of the moments.

In the data transformation stage, we parse through each event captured in the interaction log data chronologically. At each item in the log, we record around 100 researcher-authored metrics into a knowledge base. Together the metrics provide a comprehensive snapshot of the state of the programming process. For example, one metric captures the number of lines that a student pastes from a resource into their code. Other metrics require more calculations, for example evaluating if a student was editing mid-line when they stopped to look for a resource. In order to calculate these metrics, we use the log data to keep a constant state of the programming environment, including both how the programming environment is represented to the student as well as helper variables to keep track of historical values. For example, when a student goes to a resource, in order to know if they were editing mid-line when they left the IDE, we need to have a record of the last edits they made in the editor.

In the decision-making stage, we analyze the metrics at each log event to determine if any of the self-assessment moments occurred. We use two different styles of heuristic algorithms, either if-then rules when there is less ambiguity in the decision-making process (e.g. *getting simple errors*), or fuzzy logic [119] when many metrics need to be considered in parallel (e.g. *using resources to look up syntax*). For example, we use fuzzy logic to increase our confidence that a student is using a resource to look up syntax if they paste either one or two lines of code from the resource. There are a number of complexities in this stage that we considered. For example, we keep track of the previous identification of a moment in order to ensure that there are not extraneous duplicates. Another complexity is determining a particular start time for the moment as the interval when the moment can be determined may not be at the beginning. For example, we need to know how a student uses the internet to determine which using resources label is accurate. Thus, the moment will not get triggered until the student returns from using resources, but the 'start' of the moment will be logged as when the student goes to the resource initially.

As a concrete example, consider the strong indicator for the *struggling with errors* moment,

when a student runs the code in an attempt to fix a bug three times in a row. One metric for this indicator calculates whether the student is working on the same error across multiple compilations. This metric keeps track of the number of the errors in the console and the names of the errors. After each compile, we use this information (along with some additional details about code edits) to evaluate if the student is still working on the same bug.

We chose an expert system because retrospective data is time-intensive to collect. It is impractical to collect enough student-labeled data to serve as ground-truth for machine learning algorithms. Additionally, data-driven approaches often produce features that are not human-interpretable, making it difficult to understand their decisions and limitations. With an expert system, we can trace the decision process and ensure that the system is making logical choices.

5.4 Evaluation of the system

5.4.1 Methods

We evaluated our system by comparing the automatically detected moments to those manually labelled by the research team based on the codebook. We chose to use researcher labels as the ground truth because participants do not use the same diligence in labelling as researchers. While participants are able to report when events occurred, they may not report them during an interview as consistently as necessary for ground-truth data due to differing interpretations of the moments and participant attention spans. In order to gather consistent ground-truth data, we relied on the codebook that was previously verified.

We collected data from programming sessions with 33 additional students from the same university and CS2 course as our initial interviews. Again, students programmed using the Java language. By choosing students from the same course, we keep consistency in the data but can not test generalizability. The setting and procedure were the same, with the exception of the retrospective interview, which was excluded. To establish the reliability of the researcher-assigned labels, my collaborator and I independently labelled the same seven interviews, or 21% of this data set, achieving 82% agreement. We then independently labelled the remainder of the data.

One challenge in evaluating this system is establishing a way to compare moment timing between the researchers and the machine. When manually labelling the moments, the researchers picked a timeslot from non-overlapping ten-second windows (e.g., 0-10, 10-20). When comparing the system's results to the researcher-labelled set, we used an additional fifteen-second buffer on both sides of the ten-second window because the start time of a moment can be difficult to determine and might fall on the border of a window. We marked a machine detection as correct if the timestamp assigned to a label was within this forty-second window. We used a slightly larger buffer to more accurately represent two of the moments. For *changing approaches*, we used a two-minute buffer instead of a fifteen-second buffer because this moment often takes places over a few minutes, and we did not have a way to consistently identify matching start times. For *struggling with errors*, the researchers identified the start and end time for the error cycle in which the participant struggled. We deemed a system-identified label as correct if the system chose any time within the error-cycle boundaries. While both of these windows are larger, they reflect the context of these moments and the system's ability to identify these moments accurately.

After running our system on the log data from our evaluation data set, we further analyzed its performance by looking at each false positive and false negative result. The authors reviewed each case and categorized the reason for the false detection by watching the screen recording of the moment and consulting the codebook. During this process, we identified a number of instances when the researchers mislabeled moments, and also noted the limitations of our system.

5.4.2 Findings

Our results in Table 5.2 show that we had very high F1 scores for some moments, such as *getting simple errors*, and lower but still reasonable F1 scores for others, such as *writing a plan*. While precision and recall are both important, high precision matters most for interventions to ensure that real-time messages are delivered in response to true moments, and recall is most important for studies to ensure that relevant moments are not missed. The data also shows that the moments arise at varying levels of frequency; *getting simple errors* and *stopping to think* were most frequent,

Table 5.2: Results from the evaluation of the detection system.

Moment	Precision	Recall	F1 Score	Count	Human Errors
Using resources to look up syntax	82.0%	86.1%	84.0%	128	2
Using resources to research approach	66.7%	66.7%	66.7%	21	1
Changing approaches	73.1%	73.1%	73.1%	26	8
Writing a plan	60.0%	75.0%	66.7%	15	0
Getting simple errors	99.1%	97.7%	98.4%	213	13
Getting Java errors	90.3%	90.3%	90.3%	31	2
Struggling with errors	69.2%	90.0%	78.3%	26	5
Stopping to think	79.1%	75.3%	77.2%	159	15

while *writing a plan* and *using resources to research an approach* only occurred occasionally. Our system tended to perform worse for less frequent moments, likely because our codebook and system were developed using fewer observations. However, the frequency of a moment does not necessarily indicate its importance. While we do not yet know how each moment influences student self-efficacy, some of the less frequent moments may have a stronger impact on student experiences than the more frequent ones.

One benefit of our approach is that our system's decisions are transparent and can be assessed using our qualitative codebook. This enabled us to conduct an analysis on our system's false positives and false negatives. First, our analysis revealed many human errors in labeling, showing how challenging it is for humans to accurately label this type of data and highlighting the value of an automated system. Our analysis also revealed trends that provide direction for improving the system. For example, 10% of the system's incorrect decisions occurred because the researcher and system disagreed about the timing of a moment. When we designed the codebook, we focused on describing the heuristics to determine whether a moment occurred, rather than the exact start time for every moment. As a result, our system had less information to help it choose start times. Many of these moments occur over a period of multiple minutes, and therefore detection within a wider range of times could be acceptable. In the future, we would suggest either developing heuristics for determining start times during the qualitative analysis or changing the evaluation to allow the system to select any time point during the moment, as we did for *struggling with errors*.

Our analysis of the system's incorrect decisions also revealed that particular metrics were dif-

ficult to encode. For example, our system was not always able to determine when a student had resolved a particular error, which is crucial to detecting the *struggling with errors* moment. This can be quite complex, as students exhibit a wide variety of behaviors when debugging. Another challenge we encountered is that our system does not always have enough information to determine the student's purpose for using resources when it knows a *using resources* moment occurred, resulting in a lower recall for *using resources to research an approach*. Even though our metrics generally provided enough guidance for the researcher, without human intuition or contextual understanding, the system was less accurate in interpreting the variety of ways that students use resources. With more development time, we could increase the accuracy of detection for both of these moments, but it would require significant effort to fully model all potential behaviors. While it is likely not possible to fully capture the variance in student behavior in our models, our relatively high detection accuracy and our concrete ideas for improvement show that this is a viable approach.

5.5 Conclusion

In this chapter, we present a new approach for designing systems that detect student perceptions of the programming process, called *retrospective-enabled perception recognition*. We apply this approach to develop an expert system to detect programming moments that prompt students to negatively self-assess, building on expertise gained through retrospective interviews with 41 CS2 students. We evaluated our system with programming session data collected from an additional 33 CS2 students, finding that our system achieved F1 scores ranging from 66% to 98% for the eight self-assessment moments.

While we are encouraged by our system's performance, this work has a number of limitations. First, our evaluation relies on researcher-assigned labels. While we verified the labeling process through a formal qualitative analysis, researcher labels may not perfectly represent student perceptions. Additionally, while we believe the *retrospective-enabled perception recognition* approach can be applied to other problems, the system itself was built in a very particular context. We devel-

oped and tested our system with students from just one course and university, working on a similar set of problems in the same language. Additionally, all data was collected in a lab setting and not in-the-wild. Thus, the applicability of the system itself may be limited to this one particular context. From this work, we do not know if it is possible to build a system that is broadly generalizable or if different detection systems are needed based on sociocultural and course context. Additional work is needed to understand whether our system will generalize to a more naturalistic setting, more diverse problems, and other programming languages.

Through *retrospective-enabled perception recognition*, we contribute a new approach for combining qualitative methods and expert system design to detect moments that students perceive as meaningful. Furthermore, our system for detecting negative self-assessment moments has the potential to enable new studies and interventions that were not previously possible. In future work, we hope to use this system to further study how student perceptions and self-assessment impact student experiences in the programming process. This future work will continue to inform instructors and curriculum developers of CS1 programs and enable the design of real-time feedback interventions to help students re-frame self-assessment moments and improve self-efficacy.

CHAPTER 6
STUDY 4: USING ELECTRODERMAL ACTIVITY MEASUREMENTS TO UNDERSTAND NOVICE PROGRAMMER EMOTIONS

6.1 Problem

For novice programmers, code writing can be a roller-coaster of emotions, from frustration and desperation, to joy and pride [5, 120, 121, 122, 123]. Emotions are important for researchers to consider because they correlate with long-term outcomes like project and course performance [6, 124, 122, 123], self-efficacy [14, 6], and self-assessed productivity [125]. In previous chapters, we identified that some students negatively self-assess at moments throughout the programming process. While our detection algorithm can identify the moments of potential negative self-assessment, we do not know if students have reactions in those specific moments or how students experience them. Thus in this chapter, we begin to explore the relationship between emotions and self-assessment moments by identifying the events that cause students to experience emotional reactions while programming.

Identifying the specific causes of students' emotions while programming is key to improving their programming experiences. A number of studies have explored programmer emotions using a variety of methods, including asking general questions about the events that trigger emotions [126, 127], interrupting students while programming to learn about their emotions [128], and asking about affect during predetermined events [123]. While these studies provide a valuable foundation to our understanding of programmer emotions, they have a number of limitations. Participants often struggle to accurately recall their emotional reactions after the fact [28], but interrupting them as they work impacts the authenticity of the programming experience. However, we currently lack a method for analyzing emergent programmer emotions during authentic programming sessions with a moment-to-moment unit of analysis.

Measuring physiological reactions throughout a programming session may provide new insights into the emotional experiences of programmers. Emotions are not just cognitive reactions; they also create physiological changes in the body, like increases in heart rate and sweat production, which can be measured by sensors [129]. Physiological sensors provide continuous and fine-grained data that allow researchers to pinpoint particular moments when people experience emotions and analyze emotions across time. Since people may not always be aware of their emotions, physiological devices can measure emotional stimuli even when people are not cognizant of those experiences [130, 131].

With the increasing availability of sensors, cued-recall experts have suggested adding information provided by physiological data sources to retrospective interviews to assist in triggering student recall of events [132]. Physiological data provides an additional lens through which interviewers and participants can view participants' emotional experiences, which complements the cognitive and behavioral indicators that we can measure with surveys, interviews, log data, and observation [131, 129, 133]. Physiological data sources may help improve student recall in retrospective interviews by surfacing moments of high emotional responses for further discussion. However, we are not aware of any existing studies that have attempted to utilize physiological data sources with a cued-recall methodology in the computer science education domain.

To address this gap, we used electrodermal activity (EDA) sensors to capture physiological data while novice programmers work on a programming problem. We utilized this data to trigger student recall during a retrospective interview of the programming session with the goal of answering two questions: (1) What events during programming trigger novice programmers to have emotional reactions? (2) How do student remembered experiences align with visual inspection of EDA data?

6.2 Background

6.2.1 Emotions

While emotions are a common occurrence in everyday life, they are notoriously difficult to define scientifically [134]. There is agreement across theories that emotions involve multiple processes in the body, including both cognition and the work of the autonomous nervous system. This is reflected in the American Psychological Association's definition of an emotion: "a complex reaction pattern, involving experiential, behavioral, and physiological elements, by which an individual attempts to deal with a personally significant matter or event" [135]. Scherer's Component Process Model identifies five coordinated processes that make up emotions. The five processes are: cognitive appraisal of the situation, bodily symptoms, tendencies towards action, facial and vocal expression, and feelings [134]. In this dissertation, we use this definition of emotions as it summarizes research showing that emotions are multi-faceted, involving co-existing processes of cognition (e.g. thoughts), behavior (e.g. facial expression changes, activity changes) and physiology (e.g. increased heartbeat, sweating) [134].

With the recent trends towards understanding student affect and motivation, computing education researchers have increasingly studied the emotions that novice programmers experience [120, 125, 124, 126, 121, 136]. For example, Bosch & D'Mello identified novice programmer affective states, showing that novices experience emotions from engagement and happiness to disgust and frustration [123]. Kinnunen and Simon identified different aspects of freshman programmer experiences, like the 'hit by lightning' experience, which occurs when a student encounters a problem that they did not expect [5, 14].

A number of these studies have identified that student emotions correlate with measures of student success, like performance and persistence. For example, Bosch et al. identified that boredom, flow, and confusion are correlated with student performance [122]. Lishinski et al. found that emotions correlate with student performance on projects and has long- and short-term effects on course performance [6]. Studies have also found that novice programmer emotions correlate with

self-efficacy and self-assessed productivity [14, 6, 125]. For example, Kinnunen and Simon documented that after an unsuccessful programming episode, novices expressed that they had feelings of inadequacy and stupidity. Additionally, emotions likely influence student persistence through the CS major since enjoyment of programming is a main factor in student decisions to major in CS [15], and students' recollection of past programming assignments are dominated by their emotional experiences [5, 124].

6.2.2 Existing methodologies for identifying triggers of emotions during novice programming

A few studies have attempted to identify the triggers of the various emotions that students experience while working on programming problems. Two studies identified a list of triggers for both negative and positive emotions by asking novice programmers generally about the causes of their emotions [126, 127]. Despite the interview happening directly after a programming session, these studies did not direct the questions about emotions towards specific instances in that session, but were general to their programming experiences. For example, Girardi et al. asked programmers *"What are the causes for your negative emotions during programming?"* These types of questions require students to reconstruct memories based on an association, in this case recalling the programming events that aligned with experiences of negative emotions. This question format results in less accurate recollections when compared to asking students to provide detail of specific instances of a programming session [28, 29]. Drosos et al. accessed specific examples of frustration by instructing students to report their emotions on a survey throughout a programming session [137]. While this data provides specific examples of emotional triggers in-action, the programming session is not authentic because the students had to report their emotions throughout the session. Additionally, identifying the points of interests are solely reliant on students to remember to report their emotions. Although they did not directly report on the triggers for student emotions, Bosch & D'Mello addressed these issues by using a qualitative approach to understand the emotions that novices experienced during their first computer programming learning session [123]. After the participant worked on a programming problem, the researchers and participants together watched the

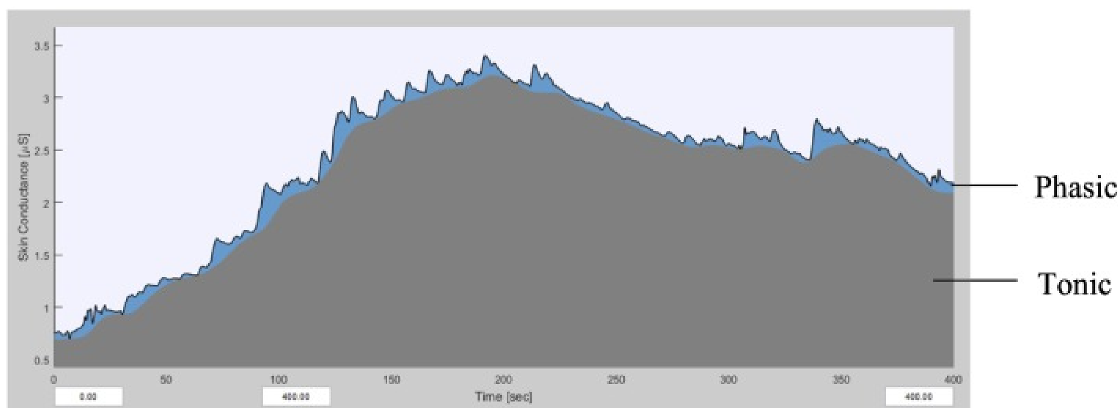


Figure 6.1: Phasic and tonic activity segmented in an EDA signal, demonstrated by Caruelle et al. [139].

screen recording and front-facing video of the session, pausing the recording at specific interaction events to ask participants about their affect. Since the probing points were predetermined, these findings may be missing instances when students have emotional reactions that the researchers did not expect. Thus, in this chapter, I suggest a new method for studying specific instances of events that trigger emotions during authentic programming sessions.

6.2.3 Physiological data analytics - electrodermal activity

When a person experiences an emotional stimulus, it creates sympathetic neuronal activity, which results in frequent, tiny changes in sweat production [129, 131, 133, 138]. These changes may not be noticeable to the individual, but can be detected through electrodermal activity (EDA) sensors. Electrodermal activity is the measurement of skin conductance, which is based on the amount of sweat; the more sweat on a person's skin, the higher electrical conductivity.

An EDA signal can be broken into two components, phasic and tonic, seen in Figure 6.1 [139]. Phasic activity is the short-term fluctuations, or peaks and valleys, that represent neuronal activity, as after a triggering event there will be an immediate spike in skin conductance. These peaks are referred to as skin conductance responses (SCR) and can be used to study temporally unfolding events [133, 131]. Interestingly, the intensity of SCRs often reflects the physiological significance of events that trigger them. Tonic activity is the general level of EDA and varies slowly, thus is

Figure 6.2: The Empatica E4 wristband [140].

referred to as the skin conductance level (SCL). While SCL is influenced by emotions, external factors can also impact SCL, like time of day [129]. Shifts in tonic level and changes to frequency and amplitudes of peaks in the phasic activity are indicators of changes in emotions [129].

Physiological data, and specifically EDA, has been used to study students' affective state during cognitive tasks [141, 142, 139, 143, 130]. EDA data addresses three common challenges with measurement of student emotions [139]:

- Emotions can occur at any time, but many data collection methods capture information at specific intervals. EDA can be measured continually through an entire activity.
- Participants do not always express the emotions they have experienced. EDA data indicates the presence of emotions, even when participants have difficulty reporting those emotions due to their inability to remember, discomfort in talking about the emotional experience, or challenges with describing emotions accurately.
- Emotions can be subconscious. Not all emotions are conscious to respondents, yet EDA can capture these sub-cognitive reactions [138].

Additionally, EDA data is simple to collect. There are many devices that can measure EDA, most

of which have a cost accessible to researchers and only require the user to wear a wrist-band or clip on their finger.

Computing education researchers have begun to use EDA as a tool to understand student emotions during the programming process. Specifically, two studies developed machine learning models that could accurately predict emotions from EDA data of a programming session [126, 127]. To create ground truth data, the researchers periodically interrupted the participants while programming to get self-reported emotions. Then, they trained and tested SVM machine learning models using the EDA data. Both studies were able to build models that could reliably predict programmer emotions. While these machine learning algorithms demonstrate the prediction power of EDA data to understand student emotions, machine learning models require large amounts of training and testing data, often for each individual student, as well as development time and knowledge. Additionally, machine learning algorithms only provide information about the outcomes of the algorithm, and not the context in which the emotions arose, the factors that determined the algorithm's decision, or the insights into the transitions between moments. Finally, the data that the algorithm references as ground-truth is not collected from an authentic programming experience as the participants were regularly interrupted.

A number of other studies have also explored the use of EDA in understanding the programming experience [144, 145, 146, 147]. For example, Wroble evaluated the number of SCRs that occurred during a programming task and found that there was a correlation between the number of SCRs in the programming session and student self-report of their overall emotion [146]. Ahonen et al. used EDA to look at emotional synchrony between pair programmers [145]. They utilized visual inspection and signal evaluation to investigate differences and similarities in pair programming roles. These prior studies demonstrate the usefulness of EDA in understanding emotions. In this chapter, we combine the benefits of EDA with retrospective interviews to further understand student emotions while programming.

6.3 Method

This study aimed to answer our two research questions by identifying the events that trigger student emotions while programming and studying whether students' remembered experiences align with EDA data. We designed a lab study in which we collected EDA data while students worked on programming problems, and then conducted a retrospective interview that leveraged the EDA data using a cued-recall technique to improve students' ability to remember their emotional experiences. We chose to analyze our data using a qualitative methodology to uncover the rich context of students' emotional experiences while programming.

6.3.1 Participants & setting

I recruited 14 undergraduate students (10 men and 4 women, aged 18-21) from a mid-sized private university in the Southeastern United States. I conducted this study at the beginning of a semester, in January 2022. All participants had only completed one introductory CS course; some participants had started a second introductory CS course. I choose this population of students because I believed they had enough experience with CS to develop perceptions and opinions, but were still deciding if they should pursue CS. I recruited students through emails sent by the professors of the introductory programming courses and announcements made in class.

6.3.2 Study procedure

I conducted two-hour interviews with participants individually, following social distancing and masking protocols. There were two sections of the interview: a programming session and a retrospective interview.

Programming session

I directed participants to work on a programming problem for 30 minutes. The problem description, which described the expected functionality and provided examples, asked participants to write

a function that removed duplicate words from a sentence, where duplicates were case-insensitive. I designed the lab study to emulate a normal programming session as much as possible. I instructed participants to use resources as they would for a normal homework assignment. Participants used their own laptops, but used the jGrasp IDE [115], which was new for many participants. When the participant began working on the problem, I left the room. While programming, participants wore an Empatica E4 wristband EDA sensor [148, 140] on each wrist, as seen in Figure 6.2. I choose to use a wristband sensor instead of fingertip sensors because it is less disturbed by the movements involved in using a keyboard and mouse.

Retrospective interview

After the programming session, I followed the procedure in Section 6.3.3 to analyze the EDA data captured by the Empatica E4 device. This analysis produced a list of timestamps of skin conductance responses (SCRs) captured in the EDA data, which indicates potential emotional responses.

I then conducted a retrospective interview [84] using cued-recall techniques [132, 149] with the SCRs as cues. The participant and I watched the screen recording and laptop camera recording of the programming session together, with the list of SCR timestamps displayed onscreen as a reference. As we watched the recording, I asked the participant to describe their programming experience and any emotions they felt. When we reached a point in the recording that aligned with an SRC timestamp, I informed the participant that an SRC had occurred and asked them to describe what happened at that moment and whether they experienced an emotion. Any time the participant identified an emotional reaction, I asked whether it was a positive or negative emotion, and what they believe caused the emotion.

One potential concern with this approach is that participants might feel pressure to create a narrative to fit the SRCs, for example due to a bias caused by the study's demand characteristics [150]. To address this issue, I discussed the nature of EDA data with participants at the beginning of the interview, before watching the recording of the programming session. Specifically, I told

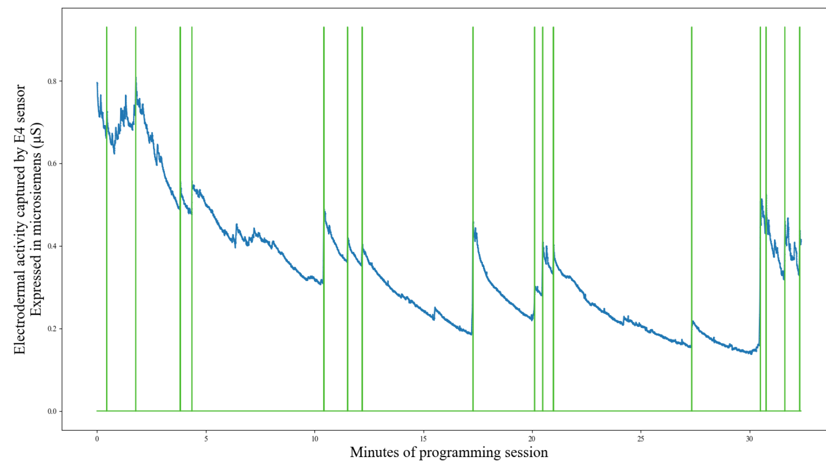


Figure 6.3: Results from skin conductance response (SCR) detection for P12. SCRs are marked by dotted vertical lines. EDA level is displayed in blue.

participants that while EDA sensors are good at detecting when emotional reactions occur, other unrelated factors can also cause SCRs, such as rapid arm movements. My goal was to ensure that participants felt comfortable sharing that they did not experience an emotion, or experienced an emotion due to a distraction rather than the programming task, at a moment when the EDA sensor picked up an SCR.

6.3.3 SCR detection

I conducted the analysis for determining the SCRs in the programming session after the participant finished programming but before the retrospective interview. To identify SCRs, I first used the peak-detection algorithm from EDA explorer [151]. To determine the correct parameters to set the tolerances for the algorithm, I conducted a parameter sweep with 16 sets of parameters. The parameters included a tolerance value, or the minimum amplitude that a peak must reach in order to be considered an SCR. I chose the parameter set that best identified distinctive peaks without capturing noise based on visual inspection of graphs of the detected peaks from each parameter set.

Finally, I visually inspected the graph to remove any mislabeled SCRs and add any unlabeled

SCRs in order to improve the accuracy of the results. Visual inspection has been used to identify SCRs in prior work [147, 139]. Visual inspection can improve the accuracy of EDA data because SCRs may not always be perfect peaks. For example, when there are multiple SCRs in a row, the second peak might start before the first resolved, making it difficult for an algorithm to identify them. The result of this analysis for participant 12 is displayed in Figure 6.3.

6.3.4 Identification of triggers of emotions

My colleagues and I conducted a qualitative analysis of the interview data to identify types of moments in the programming session that triggered emotions. I generated an initial codebook for moments that trigger emotions. I started by reviewing a subset of the interview transcripts with an open-coding protocol [152], focusing on instances when participants expressed an emotion and described a trigger for that emotion. I iterated on the themes as I reviewed more transcripts.

This process generated a list of emotion triggers and associated descriptions of the triggers broken into positive and negative emotion categories. For example, after having an issue with the compiler a participant said: *"I think I was happy that ran"*. We categorized the trigger of the positive emotion in that moment as "resolving interface issues".

My colleague and I then used the codebook to independently code emotion-trigger pairs in the data. We coded interviews separately and discussed discrepancies after coding each transcript, iterating on the codebook when necessary. To check consistency, my colleague and I independently coded three transcripts, or 21% of the data. We had a percent agreement of 83%, which represents good agreement. I coded the remainder of the transcripts.

6.3.5 Analysis of EDA data with respect to student experiences

I next investigated how the EDA data reflects student experiences throughout the programming session to better understand how EDA data can be used to interpret the student programming process. I aligned the graphical representation of the EDA data with participants' descriptions of their experiences across the entire programming session. While in the interview I only used the

timestamps of SCRs, for this analysis I considered three features to reflect more gradual emotional changes that take place across time. The three features that we used to analyze the EDA graphs are: the amplitude of SCRs, frequency of SCRs, and drifts or changes in skin conductance level (SCL), as they are most indicative of emotions [129].

During the analysis, I annotated each participant's EDA graph with the student's recollections of the activities they performed and the emotions they experienced during the programming session. This side-by-side view of student report and physiological data facilitated analysis of how EDA evolved with students' emotional reactions as they worked through the problem. Then I wrote memos about the patterns that she observed in the EDA data, observing how the descriptions of the events aligned with changes in the EDA features [153]. The memos were used to detail my observations of the notable behaviors of the EDA data as related to the student recalled experiences. Finally, I reviewed the memos and the annotations to identify common patterns seen across multiple participants.

6.4 Findings

We share our findings, corresponding to our two research questions, in the following two sections. First, we present a set of events that trigger students to experience positive and negative emotions while programming and present detailed examples from our data set. Second, we present evidence of the relationship between student remembered experiences and patterns in their EDA data, identifying a few common emotional patterns that arose across multiple participants.

6.4.1 Events that trigger student emotions

To answer our first research question, we identified a set of programming events that students said triggered an emotional reaction. From the qualitative analysis, we identified 21 themes of these events, including 8 that participants reported caused positive emotions (see Table 6.1) and 13 that participants reported caused negative emotions (see Table 6.2). When identifying the triggers, we only labeled moments where both the emotion and trigger were present. Thus, there may be other

Table 6.1: Events that triggered positive emotions during the programming session.

Trigger	Explanation	Count
Getting direction from a resource	When a student learns something or finds something useful in a resource.	22
Typing code / making progress	When a student has positive emotions from the action of typing code in the editor. This occurs when there is the feeling of being productive.	11
Completing a step in coding problem	When a student completes a task, whether it is a step in the programming problem, or a subgoal in their process to completing the problem. This is outcome oriented and denotes concrete completions of a step, generally evidenced by running code, but not always.	10
Having a plan	When a student has a plan for their code. This also could be relief or excitement from recognizing a new plan.	9
Fixing errors	When existing errors are fixed and no longer present.	7
Remembering syntax correctly	When a student remembers syntax without external help. This is not about writing code, but specifically remembering without help.	5
Understanding the problem statement	When a student has positive reaction from reading the problem statement because they understand the question.	3
Resolving interface issues	When a student resolves an issue unrelated to solving the programming problem. Some examples include: IDE, finder, resource access. This is the end of the "interface issues" trigger.	3
Other	When a student experiences a positive emotion but the triggering event does not fit into one of the specified labels.	4

Table 6.2: Events that triggered negative emotions during the programming session.

Trigger	Explanation	Count
Not knowing something	When a student feels they need to use a resource because they do not know something, because they forgot it or do not know it.	23
Interface issues	When a student has issues unrelated to solving the programming problem. Some examples include: ads, IDE, finder, resource access.	15
Resource not helping	When a student uses a resource but does not find it to be helpful. This may occur because they do not understand the resource or can not find the right source.	15
Realizing there is an error	When a student gets an error. This is about the existence of an error and not the struggle with fixing the error	14
Struggling while trying to fix an error	When a student struggles with or spends a long time working on errors, whether they are simple or not. For example, repeatedly trying to solve the same error multiple times unsuccessfully.	12
Intimidation from reading problem statement	When a student has a negative reaction to reading the problem, either because they do not know how to approach the problem or because they are intimidated by it.	11
Not making progress	When a student feels they are not making progress towards the solution. This could be displeasure about spending time thinking or not making progress on the problem.	6
Realizing code/plan not working as expected	When a student has implemented code and realizes the code or their plan for the code is not working in the manner they expected. This occur when running or from reading the code.	4
Changing approach / deleting code	When a student changes the approach they have been taking towards the problem. This often appears when a student deletes lines of written code.	4
Not understanding error message	When a student does not understand the text of an error message.	2
Struggling to fix program behavior	When a student struggles to fix a logic error or incorrect code behavior.	2
Encountering code formatting issues	When a student has issues with the formatting of the code, not the functionality of the code.	2
Not remembering problem description	When a student has to read the problem statement again because they forgot it or misunderstood it.	2
Other	When a student experiences a negative emotion but the triggering event does not fit into one of the specified labels.	5

instances of these triggers where participants did not experience an emotion. The count represents the number of instance when these triggers caused an emotion. Interestingly, many of the positive events were opposites or counterparts to negative events. For example, we found that negative emotions occurred when a student experienced issues working with their tools (code: “interface issues”), and positive emotions occurred when a student relieved those issues (code: “resolving interface issues”).

Many of these moments involved interaction with a variety of information, resources and tools, like reading the problem statement, interacting with the console in the IDE, and searching for resources online. While students may interact with the same tool or process, we often found that they experienced different emotions depending on their intentions, responses, and results of the interaction. For example, students often searched for resources to help them complete the coding problem, but their emotional responses differed depending on whether they found the resource helpful (code: “useful resource”), unhelpful (code: “resource not helping”), or difficult to use (code: “interface issues”). We also found that students had different reactions to the exact same circumstances. For example, some students had a negative emotional reaction when they could not remember syntax and used a resource to look it up (code: “not knowing something”). However, other students in the same situation did not experience an emotional reaction at all.

While we did not reference the physiological data while conducting this analysis, that data informed the retrospective interview. One risk of this approach is that participants may have felt a need to explain the SRCs, even if they did not experience emotions at those moments, which could lead to inaccurate trigger-emotion pairs. To confirm that participants felt comfortable sharing that they did not experience an emotion, or experienced an emotion due to a distraction rather than the programming task, we reviewed all instances where I asked the participant if they experienced an emotion when an SRC was detected. We found that all 14 participants said that they did not experience an emotion at least one time when directly asked. For example, when asked if he experienced any emotions at a particular SRC, P3 said *“I think not yet. I’m not sure what it would be yet.”* Many students also shared that external factors impacted their programming session.

For example, when asked if he experienced an emotion at a particular point in the programming session, P10 said *"No, I don't think so. I just started texting on my phone."* These responses provide some confidence that students were accurately reporting their emotional experiences in the interviews.

The benefit of this methodology is that it allowed us to question students about very specific moments during the programming session. As a result, we were able to dig into the specific experiences that led to emotions. In the following sections, we describe nuances and overarching themes for some of our most frequent codes, highlighting differences in the ways that students described the triggers of their emotions.

Not knowing something

"Not knowing something" was the most frequent trigger for negative emotions in our data set, with 23 instances across 11 of our 14 participants (79%). The emotions that students most frequently described at these moments were frustration (7 mentions) and annoyance or irritation (8 mentions). Some students also associated this moment with embarrassment or shame (4 mentions).

When describing the moments of "not knowing something", participants frequently mentioned that they had once learned the content they were trying to remember. *"I know I've written this type of code a lot of times. So I'd say I was probably annoyed with myself for not remembering it...that it didn't come in my head straight away"* said P3, when describing his annoyance for not remembering the syntax for using the length method. P10 described his lack of memory for how to separate words in a String, saying *"It was annoying because I had done that before, I just couldn't remember"*. The memory that this information was once at their fingertips seemed to contribute to their irritation that it was now gone.

Participants also described the simplicity of what they had forgotten as a reason for their negative emotion around forgetting. *"I was a little bit ashamed that I forgot how to do something so simple in Java,"* said P11, describing her memory lapse when importing the Java Scanner. P5 expressed that he "should" recall the typical first line in a Java program: *"[I]t was muscle memory*

back when I took that class, where I would just import `java.util`*, and then I forgot how to spell it."*. The participants perceived these minor details as basic, and thus within their capacity to memorize.

At times, participants generalized their lack of knowing something quite broadly, stating that they had forgotten everything about a topic or even a whole course. P1 said, *"most of my knowledge from the class I might've forgotten, so going back and having to use a resource was a little bit irritating, certainly."* P13 recalled feeling "embarrassed" after watching the screen recording of getting errors in her code. She reflected, *"I realize[d] I don't remember any of this"*. Upon looking at her previous assignments, P8 described her emotional response: *"okay, this is harder than I thought. I don't really remember anything from past semester. So I would say again, some panic"*. It is unlikely that these students forgot all of the course content, but in their emotional state they felt a substantial lack of knowledge.

In each of these instances of the "not knowing something" code, we saw that self-judgment was present along with the negative emotional response. When these participants did not know something or had to look something up, a negative assessment of their own ability was often a key factor in their emotional appraisal of the moment. This aligns with past studies that found correlations between self-efficacy and negative emotions [14, 6]. This also aligns with past work by Gorson & O'Rourke that found that students negatively self-assessed when they needed to use a resource to look up syntax or research an approach [31].

We also found that students often had the opposite reaction when they did know something. Specifically, we found five instances where students experienced a positive emotion after "remembering syntax correctly". For example, after P3 wrote the initial structure of his code, he said *"I was pretty happy that I did remember it on the first try, which was cool."* This indicates that student emotions are often tied to whether or not they can recall coding content while programming.

Getting direction from a resource

"Getting direction from a resource" was the most frequent trigger for positive emotions in our data, with 22 occurrences across 11 of our 14 participants (79%). When participants mentioned feeling a specific emotion after finding a resource helpful, they were most likely to describe relief or relaxation (6 mentions), happiness or joy (4 mentions), and excitement or hope (3 mentions).

We found that sometimes, simply the realization that they had access to a helpful resource was enough to trigger a positive emotional reaction. P8 found relief when she realized she could search the internet for help despite not yet identifying specific content to use in the program at hand. She said: *"It's like, oh, okay. Yes. I can use Google. I think it's a sense of relief, like okay, I have more sources. I can use other people's ideas. I just remembered that."* Similarly, P2 recalled that his ZyBooks content might be a useful place to find insight on the problem, leading to happiness. He described the triggering moment as, *"thinking about going onto my class content, which I guess to some extent it relieved me from me being lost."* In both cases, the possibility of finding something helpful in a resource was enough to spark positive emotions.

Participants often had a specific goal in mind when looking through resources, and thus experienced positive emotions when they found it. P4 described such a scenario as she looked through past programs on her computer to find out how to use the `.indexOf()` method: *"I knew what I wanted, but I didn't know the vocabulary for that and I just found it on my notes."* P14 reported a similar situation, saying she felt *"a little bit of excitement that I saw a result that was what I was looking for"* as she searched Google for information about how to split Strings.

Other times, participants looked through resources in a less targeted manner, and thus when they chanced upon information they deemed helpful they experienced a positive emotion. After P12 typed a broad Google search, he recalled a moment of joy. He said: *"this `.split()` method did not come to me before, in my mind. And now I just saw it randomly. So I thought maybe this would work."* P8 had a similar experience when using Google. She recalled positive emotions when looking through a Stack Overflow page: *"I saw `.length` and my thought process was of course you can do it with that `.length`."* These participants had a chance encounter with a useful

clue, resulting in a positive feeling.

We also saw the opposite reaction to occur when students were not able to find the help they needed from a resource. We found 16 instances of negative emotions when participants encountered moments of “resource not helping”. For example, when P1 was not able to use a website to solve his confusion, he stated *“this one irritated me so much, just opening this website. I was not having it. It was no help ... I was annoyed.”* P13 had a similar experience, stating that he was frustrated when he was *“not finding what I was looking for.”* These findings suggest that resources can often be a source of emotional stimulus depending on if students are able to find the information they need.

Typing code / making progress

“Typing code / making progress” appeared as a trigger for positive emotions in 11 instances across 6 of our 14 participants. Participants expressed happiness or general positivity in these moments (6 instances), as well as a sense of accomplishment or confidence (2 instances) and relief (2 instances). Participants described these moments in terms like “making progress” (P2, P14), “moment of progression” (P12), and “getting/going somewhere” (P1, P8, P14).

When asked about why they had a positive emotion in these particular moments, participants often described the process of writing code. For example, when P2 was recalling a moment when he erased erroneous code partway through the coding session, he said, *”I pushed delete and I’m thinking of actually progressing”*. He described his emotion in that moment as *”happy that I’m starting to write something”*. For P14, her feeling of hopefulness was triggered by *”seeing the program start to come together and adding more things to it”*. P12 described the precise moment when he felt positively about his progress as when he typed code in his editor: *”When I start writing* `int i` *and* `int x` *is equals to one, for this length, I feel a moment of progression, that I’m actually going forward. I’m actually going forward in this problem towards the solution”*. These participants associated the action of adding content to the editor or the anticipation of typing code with a positive sense of progression, even though they had not yet run their code to test

whether it truly worked.

The opposite was also true: negative emotions were associated with “not making progress”, which were often tied to not writing code or typing. “Not making progress” was a trigger for negative emotion for 4 participants, across 6 instances. Participants used a variety of terms to describe their feelings in this scenario, including ”anxious”, ”worry”, ”frustration”, ”stress”, and ”distress”.

P1 expressed that spending too much time thinking and not starting to code soon after reading the problem statement led to a feeling of ”anxiety”. He said: *”I was processing it and I realized I was taking a little bit too long to start coding. So I was like, ‘Shoot. Okay, let’s go. Let’s transfer over to start coding’”*. Even in the beginning of the programming session, P1 felt negatively if he was not implementing code.

At times, the emotions around making progress were so strong, they determined the participant’s behavior. For example, P2 was using the internet to get help on the problem, but did not find the answer he was looking for. He felt frustrated *”because I’m not actually progressing in what I want to do”*. Instead of continuing to use resources, he went back to his code and started typing. He said,

> *I know one thing that I did wrong, which is just go back to the code. I should have known that I’m not going to just figure out suddenly something off of just going right back into the code and doing something. I just really felt like I wanted to progress...I wanted to just I guess, still maybe delete something or maybe type something, but it’s not really even useful. So, I just wanted in any way, having the feeling of progressing.*

His desire to have the positive feeling of making progress was so strong that he stopped and went back to the code, even though he admits that it was futile.

We found that our participants’ sense of progress can drive emotion, both positively and negatively. This sense of progression is often associated with the action of typing code in the editor, whether it is beneficial or not and can drive participant programming behaviors.

Realizing there is an error

We found that the moment when participants realize that an error is present can yield negative emotions. We identified 14 of these instances across 7 of our 14 participants (50%). These participants expressed specific emotions like frustration or annoyance (6 mentions), worry (2 mentions), and disappointment (2 mentions) upon realizing there was an error in their code.

Participants shared that one reason ”realizing there is an error” can trigger emotions is that it revealed that their code was not correct. ”Realizing there is an error” was more likely to cause negative emotions when participants expected their code to be successful, versus when they did not expect it to work. P1 described such a moment: *“The main thing about the error that frustrated me was that I thought I had a solution... but I just didn’t”*. Similarly, P7 described his disappointment at seeing an error message after compiling his code as: *“It’s just a let down, as I said. It’s a confidence roller coaster. Hitting that compile button, I knew that it wasn’t going to be completely correct, but yet I had the confidence it was”*. Although he was aware that it was unlikely that his solution would work, he still believed it would, and thus was saddened by the error.

Even when the error was relatively minor and easy to fix, some participants still felt negative emotions. For P6, a minor error caused a negative emotion because it was a repeated error. He described his reaction as: *“I got the same error I got before. So I knew how to fix it, but then I had another error, but it was something really simple. So probably just another slight frustration”*. While the frustration was slight, P1 had a negative emotion after experiencing multiple errors in a row. P5 had a similar experience, where he felt annoyed at a small error after receiving multiple, even though the error itself was understandable and easy to fix. He said, *“I guess that would’ve been an instance of having screwed that up a couple of times and then again, making another tiny little mistake”*.

Just the recognition that an error exists caused a negative reaction for some of our participants. At times, these errors indicated large issues in their plan or code, while on other occasions, the errors were quite simple. This may be due to the circumstances in the programming session, like finding multiple errors in a row, or negative views on errors.

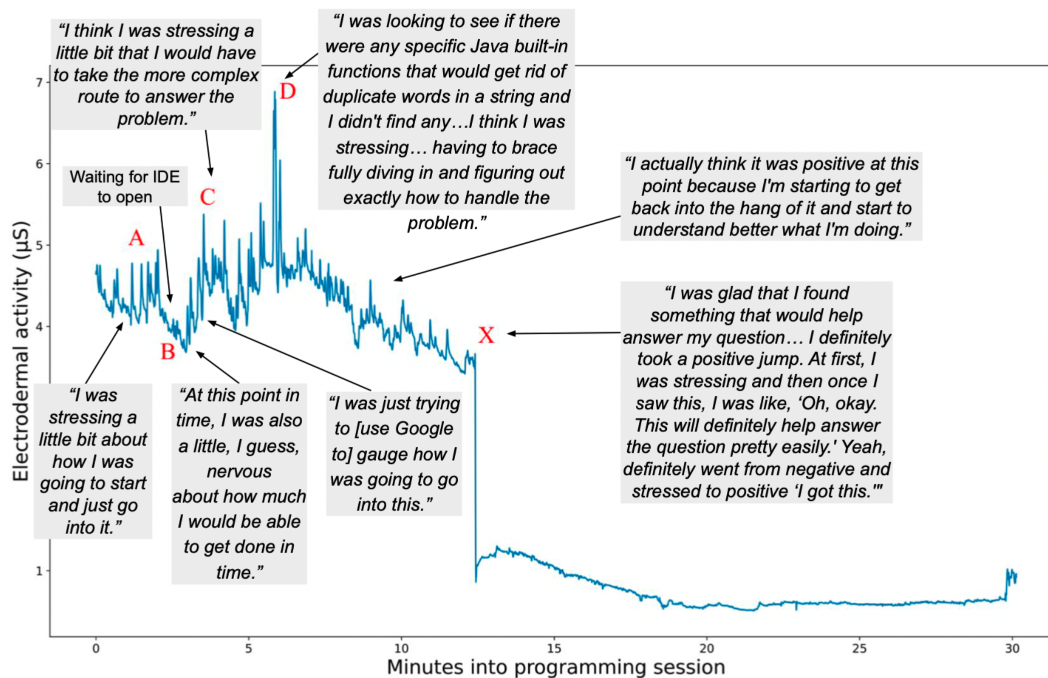
6.4.2 Emotional experiences reflected in EDA data

To answer our second research question, we explored the relationship between the participants' remembered experiences and the tonic and phasic changes in the EDA data, finding strong alignment. Specifically, the EDA data provides a map of student experiences, not only highlighting emotional moments (reflected by peaks), but also aligning with their emotional state over time (reflected in tonic level and peak frequency and amplitude). We share two participants' programming episodes to demonstrate how EDA data reflects their experiences. We also share three patterns that we observed across participants in the analysis process.

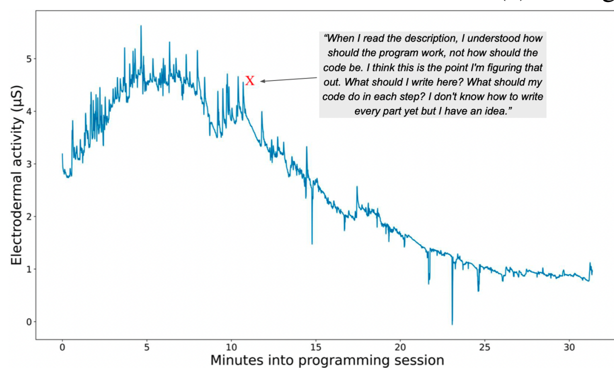
Participant 11

We start by describing the programming session of Participant 11 (P11) and the aligning EDA data. See Figure 6.4a for the respective EDA graph with associated quotes. At the beginning, P11 described feeling nervous and stressed about working on the problem. This initial reaction to the programming problem aligns with the small but noticeable peaks that we see around **Marker A**. Following **Marker A**, we observe a downward slope in the EDA data at **Marker B**. At this point, P11 is waiting for a resource to open. From the interview data alone, we would not know if P11 continued to feel stressed during the waiting time or was just sedentary. From the reduction in peaks and downward slope of skin conductance level (SCL) in the EDA data, we can infer that the participant was sedentary in this transition. Once P11 has access to the resource, she reported feeling nervous and anxious about how much she will complete. This nervousness occurred around **Marker C**, when the SCL rose and the peaks increased.

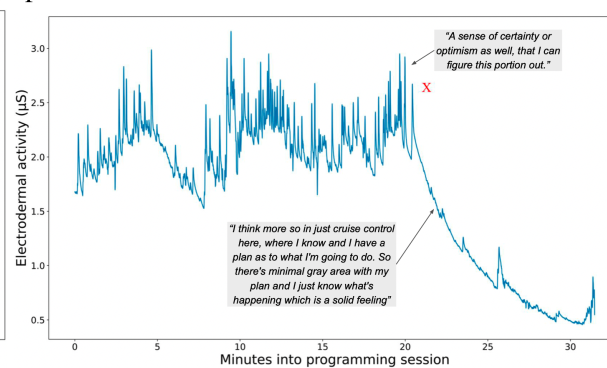
After searching through resources, P11 realized that the preset function she was looking for did not exist, and she would have to write the code for this sub-task herself. This caused a strong emotional reaction, which lines up with the large spike that can be seen at **Marker D**. Afterwards, she continued to look for a resource to help solve the problem. The SCL drifted down as she made progress in understanding the problem and determined an approach. Around the time of **Marker X**, she found a website that helped her develop a concrete plan for implementation. After that



(a) EDA graph of P11



(b) EDA graph of P8



(c) EDA graph of P9

Figure 6.4: EDA graphs of participants demonstrating the ”Cruise Control” pattern, which begins on each graph at the Marker X.

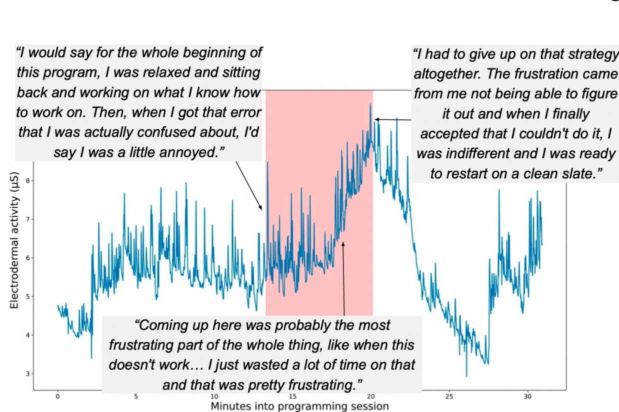
point, she was much calmer. Her energy shifted from determining how to solve the problem to implementing the plan laid out in the resource.

"Cruise Control"

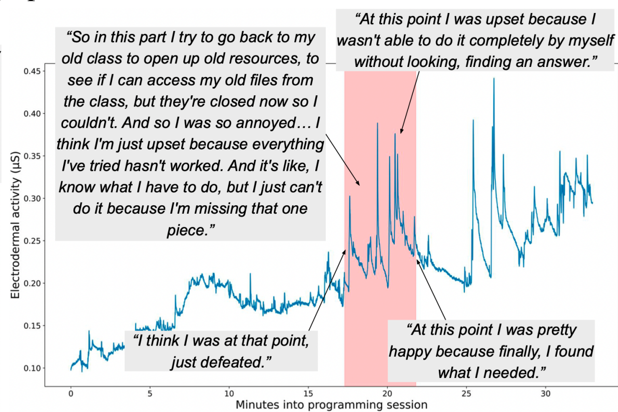
For a number of participants, we found that a large decrease in EDA (both SCL and phasic activity) occurred at the same time as the participant determined their plan and started to implement their code. This occurred for P11 at **Marker X**, as described in the previous section. We also saw this phenomena with P8 and P9, shown in Figures 6.4b and 6.4c. These participants described an event that helped them determine their plan for approaching the problem, allowing them to shift their focus from research and planning to implementing the new approach. The participants described feeling calmer and less stressed while implementing compared to planning, because they perceive implementing as a more confined problem space then planning. P8 described why she was less stressed after making a plan: *"the first part was more frustrating and the second part was more exciting than stressful because as time passed, I had a better understanding of the problem and better understanding of how to use the resources and got comfortable with the situation more"*.

The EDA data for this point of the programming session demonstrated very strong similarities across the three participants. At each of these moments, there was a steady, large drop in EDA followed by consistently lower SCL and phasic activity. This can be seen in Figure 6.4, where we have marked an X at this moment in each of the graphs for P8, P9 and P11. P9 used the term "Cruise Control" to describe the implementation phase. He said: *"Yeah, I think more so in just cruise control here, where I know and I have a plan as to what I'm going to do. So there's minimal gray area with my plan and I just know what's happening, which is a solid feeling."* His description of "Cruise Control" aligns with the other students' experiences, as they all expressed that they felt that the problem was mostly solved after the turning point, and was evidenced in the EDA data.

(a) EDA graph of P1



(b) EDA graph of P3



(c) EDA graph of P10

Figure 6.5: EDA graphs demonstrating high emotion sections, indicated by the shading.

Participant 1

Next, we describe P1's programming session and respective EDA data. See Figure 6.5a for the respective EDA graph with associated quotes. Shortly after beginning, P1 reported being disappointed when he realized that he was spending a long time thinking and had not begun to code. He expressed that he was anxious and thought *"Shoot. Ok, let's go. Let's transfer over to start coding"*. This feeling came with a *"sense of urgency"* to start implementing, which aligned with a peak in EDA at **Marker A**. Later in the programming session, P1 tested his code and got an error. The error aligns with the peak in EDA at **Marker B**. After the error, the annoyance continued as the participant could not determine how to fix the code despite using resources. This aligns with the section following **Marker B** where there continues to be frequent peaks in the EDA data. He then recognized that he was unsuccessful at using resources around **Marker C**, expressing disappointment that aligns with a time when the EDA peaks have high amplitude and are close together. Both the density and height of the peaks and the participant's qualitative description indicate that this shaded section was a strongly emotional and frustrating part of the session.

P1's SCL drifts downward as his emotions became more positive. He shares that he actively calmed himself down. He describes that he was *"attempting to gather my thoughts and bring my emotions down to a level where I could actually make good and viable steps towards working on a solution."* Then, he recalled feeling calmer and starting to make progress around **Marker D**, where there is a decrease in SCL and less frequent peaks. The negative emotions reemerged at the end of the session. He described his thinking as: *"being reflective, but in a bad way. More as in, 'Wow. Well, I suck'"*. This negative emotion and evaluation aligned with **Marker E** on the EDA graph, which shows a large, rapid increase in EDA followed by a few additional peaks.

High emotion sections of a programming session

A visual inspection of P1's EDA graph shows an increase in physiological activity around Markers B and C with all three indicators of physiological reactions: frequent peaks, increased altitude of peaks, and increased SCL. As expected, this aligns with his description of particularly strong emo-

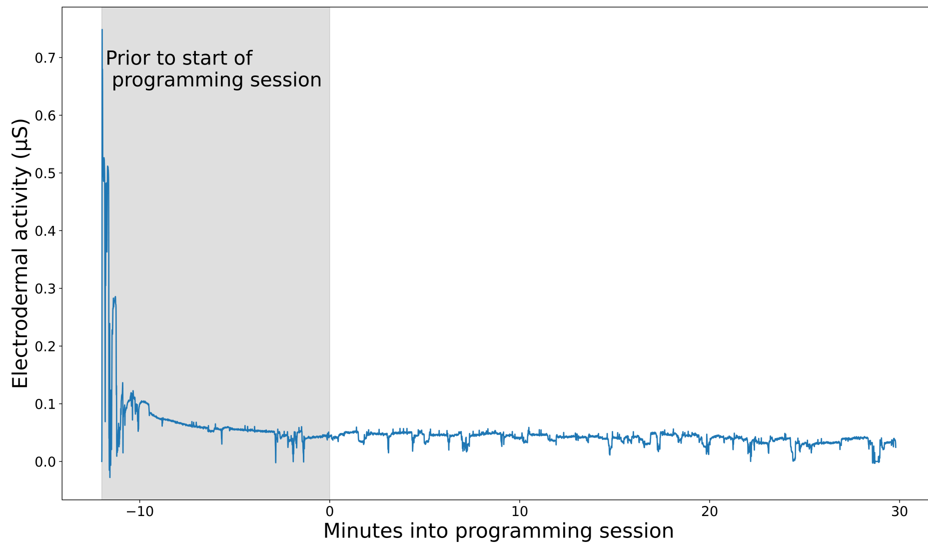
tions, indicating that he reached a peak of frustration. Looking across the EDA graphs, we found other participants with a section of the programming session that has multiple indicators of physiological reactions and aligns with a particularly emotional section of the programming session, like P3 and P10, which can be seen in Figure 6.5. These sections may be particularly important to identify and understand because participants are having such a significant physiological reaction to the programming experience in these moments.

P10 had a section of high frustration and annoyance because he did not have the resources he wanted and was not able to remember how to do part of the programming problem. This frustration directly aligned with peaks in the EDA that had significantly higher amplitudes than the earlier section of the problem and slight increase in SCL, seen at the beginning of the shaded section of the graph. The peaks persisted as P10 could not find what he was looking for. Relief from this highly emotional part came when he found a resource. The peaks subside in the EDA data and the SCL began to lower at this time, right after the shaded part of the graph. In the identified part of the programming session, P10 experienced strong emotions, aligning with high amplitude and frequent peaks in the EDA data.

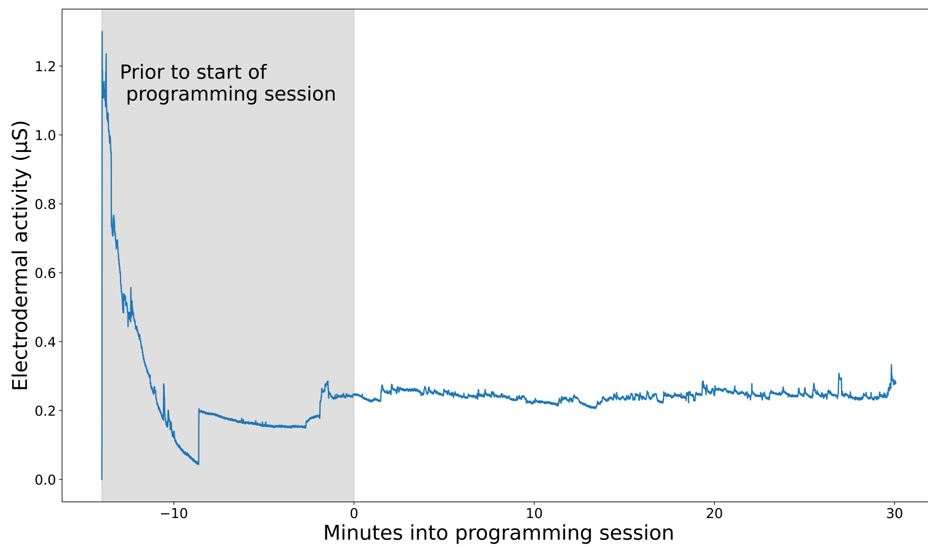
In the EDA graph of P3, there is an increase in frequency and amplitude of the peaks (at the first part of the shaded section) followed by an increase in SCL (at the second part of the shaded section), which aligned with a particularly frustrating part of the programming session as P3 struggled with an error. Following that initial error, the participant continued to struggle with errors. Around the end of the shaded part, the participant was feeling even more frustrated. He described this as: *"I'd say that coming up here was probably the most frustrating part of the whole thing, like when this doesn't work."* At the time when there was an increase in SCL and high peak amplitude and frequency, P3 describes the highest frustration.

Low EDA activity indicating fewer negative emotions

While the majority of the participants had both tonic and phasic activity in their EDA data throughout the programming session, two participants, P4 and P14, had steady EDA levels throughout the



(a) EDA graph of P4



(b) EDA graph of P14

Figure 6.6: EDA graphs demonstrating participants who experienced few negative emotions while working on the programming problem.

entire programming session. Specifically, the graphs for P4 and P14, seen in Figure 6.6, show consistent EDA during the programming problem, both in SCL and in the lack of phasic activity. For both participants, this sharply contrasts their EDA activity before the programming session, in the shaded region, which starts with high SCL and has phasic activity, demonstrating the range their EDA could reach if they encountered emotional stimuli.

During the qualitative coding analysis, we identified the fewest instances of negative emotions for these two participants. P4 and P14 had 2 and 4 negative emotions respectively, while the remainder of the participants had an average of 9.25 negative emotions, ranging from 6-15. P4 explained why she did not get an emotion when she realized there were errors, a trigger for some of the other participants. She said: *"Every time after I type code and I run it for the first time, I expect it to fail. So that's why ... it didn't affect me that much either way."* Similarly, when asked about emotions, P14 expressed that he was mostly thinking and not experiencing emotions. For example, after running his code, he said *"I think it's still mostly just kind of thinking and sort of analyzing, but the program just outputs. Trying to think about what to do next."* Interestingly, both P4 and P14 had similar number of positive emotions to the rest of the participants, 6 and 9 respectively, while the remainder of the participants had an average of 5 positive emotions, ranging from 0-9. In our data, positive emotions seem to not be as influential to phasic or tonic activity in EDA as negative emotions.

6.5 Discussion

Our analysis of retrospective interviews informed by EDA data from 14 undergraduate students allowed us to identify 21 different events that trigger emotional experiences while programming. These triggers and associated emotions provide new insights into the programming experiences that are most salient for students. The most common event to trigger a negative emotion was the experience of not knowing something, highlighting the pressure that students put on themselves to remember syntax and problem-solving approaches they have used previously. Similarly, the most common event to trigger a positive emotion was getting direction from a resources, showing

that finding direction and having a plan is both relieving and exciting. Interestingly, many of the triggers of positive and negative emotions were opposites of each other, which can help us identify the big moments that are likely to produce emotional reactions, such as making or not making progress and getting or fixing an error. While all but one participant experienced positive emotions, all participants experienced negative emotions and we saw a larger number of negative emotions overall. Furthermore, many participants exhibited strong reactions to negative triggers, for example by jumping to self-judgment, which may suggest underlying issues in programming confidence.

These insights into students' emotional experiences while programming have important implications for research and practice. Previous research shows that experiencing negative emotions while programming correlates with lower project and course performance, both in the short and long term [21]. As a result, the finding that many students experience more negative emotions than positive ones while programming is problematic, particularly since some students show indications of negatively self-assessing in response to emotionally-triggering events. Future research could explore the relationship between emotional reactions and other key factors, such as self-efficacy [14, 6], sense of belonging [7], and negative self-assessment moments [31] to better understand the implications of negative emotions while programming. In the near-term, our findings on the common triggers of both positive and negative emotions could inform pedagogy and practice, for example by prompting discussions about common emotional experiences and discussing strategies for managing feelings of frustration, anxiety, or under-confidence while programming.

Our second analysis identified broader patterns in the EDA data that aligned with participants' remembered experiences. We found that the EDA data clearly reflected the narratives that students described in their retrospective interviews, even though participants did not have access to the full EDA graphs. Common patterns, such as a reduction in tonic SCL in the EDA data after devising a plan and beginning to implement it, arose from the data despite our small sample. Furthermore, we found that students can have very different emotional experiences while programming. Some students experience very few emotions, reflected in their steady and low EDA data, while others

experience periods of strong and intense (usually negative) emotions, which aligned with high phasic activity and a raise in SCL. These initial findings suggest many directions for future research. For example, researchers could use patterns in EDA data to compare student experiences or isolate particularly interesting segments of a programming session. Further studies could also use this methodology with a larger sample of students to identify additional common patterns in student programming experiences, and potentially study the relationship between these patterns and outcomes like performance and self-efficacy.

This research was enabled by a new methodology that combines EDA data with retrospective interviews. Our findings suggest that EDA data can serve as a valuable resource for prompting student recollections during interviews. Students easily identified and described the emotions they experienced at SCRs, along with the associated context in their programming session. They also felt comfortable telling me when no emotion had occurred at an EDA peak, suggesting that their recollections of emotions were valid. As a result, we believe this is a promising method for gaining insights into students' emotions while programming. In the future, a similar approach could be used to explore emotional reactions during many types of coding activities such as reading unfamiliar code, using instructional materials, or pair programming. Future work should also further validate this approach by using an experimental design to compare traditional interviews, retrospective interviews that use only screen captures of programming sessions, and retrospective interviews with screen captures and EDA data. This would allow us to directly measure the added value of EDA data in prompting detailed student recollections of their emotional experiences.

6.6 Limitations

While this study contributes new insights and an approach for understanding student emotions, it has a few important limitations. First, we recruited students from a single private university. This limits the generalizability of our findings, because students in a different learning environment might have a different set of emotional triggers or might exhibit different physiological reactions to a programming session. In addition, while our algorithms and methods were designed to reduce

the impact of noisy data, there is always the potential for data inaccuracies when using physical sensors, resulting in either false-positives or false-negatives in the EDA peak data. Furthermore, while we incorporated physiological data into our interview protocol with the goal of improving recall of emotions, we are still limited by participants' ability to identify and describe those emotions. Since we provided participants with a list of SCRs during the interview, one concern is that participants may have felt pressure to produce a narrative that explained the peaks. Our data suggests that participants felt comfortable notifying us when no emotion occurred at a given peak, but it is still possible that the presence of the peak data overly influenced student recall of the events that triggered emotions. Conducting a formal study comparing retrospective interviews with and without EDA peak data could help further validate the reliability of this approach for investigating emotional experiences in the future.

Since us as researchers are a tool in the analysis, the unconscious bias that we have from our previous experiences and research may impact the qualitative analysis. This potential limitation is generally unavoidable as we are human, however we designed the methods and procedures to limit this concern. For example, we used the EDA data to guide the probing in the interview so that I did not have to determine which moments to ask about. During the analysis, we looked for instances where the participant expressed an emotion and used that to guide the moments that we included in the analysis. Finally, we used IRR in order to confirm that we were consistent and reliable in our interpretation of participant talk. These considerations provide confidence that we addressed the unconscious bias as much as possible.

6.7 Conclusion

In this chapter, we leverage electrodermal activity (EDA) data to prompt student recollections of their emotional experiences while programming during retrospective interviews. This approach allowed us to identify 21 distinct events that trigger positive and negative emotions during programming sessions, providing new information about the experiences that are most salient for students. We also showed that there is a strong relationship between student remembered experi-

ences and the tonic and phasic elements of EDA graphs, demonstrating the expressiveness of EDA data. Our findings suggest that many students experience more negative than positive emotions while programming, and that different students can have very different emotional reactions to the same programming events. This research opens up a number of potential areas for future work, including studies of the relationship between emotions while programming and other factors such as self-efficacy or self-assessments as well as further investigations into the utility of EDA data for prompting student recollections of emotions. We hope that this research inspires a continued focus on the emotional experience of introductory programming students.

CHAPTER 7
DISSERTATION CONCLUSION

I present a number of contributions in my dissertation that improve our understanding of student experiences working on programming problems, specifically focusing on student perception of intelligence. The contributions from my dissertation can be broken into four categories: practical, conceptual, methodological, and technological. In the next few paragraphs, I discuss the contributions from each of these categories.

The findings from my research studies will inform how practitioners coach students, both in computing education classes and informal learning spaces. Specifically, practitioners can utilize the insights into the moments during the programming process that cause students to make negative self-assessments and experience emotional reactions. For example, with the list of criteria that students use for self-assessments, practitioners can adjust their lectures or grading rubrics to encourage students to have more productive perspectives of programming ability. When witnessing students experience these moments, practitioners can also provide encouragement to deter false perspectives of the programming process. Additionally, practitioners can help students address self-critical biases by increasing awareness of the bias and encouraging students to evaluate themselves fairly. Finally, our findings indicate that some students have inaccurate perceptions of professional programmers, suggesting that practitioners should provide more opportunities for students to learn about professional programming practice to promote more accurate perspectives on how professionals work.

In this dissertation, I make conceptual contributions that provide new frameworks for understanding novice programming experiences. Specifically, we started with focusing on growth mindset theory as a framework for understanding how students think about their programming intelligence while working on programming problems. Through our research studies, I have found that understanding student perspectives on programming intelligence is much more complicated than

mindset theory allows. I found that students use widely varying criteria for evaluating their programming intelligence. In addition, many of these criteria are more critical than the criteria that professionals and instructors use for evaluating programming process. While there is some indication that students use unnecessarily strict criteria in prior studies, no study identified a set of these criteria or documented their importance to student perceptions and self-efficacy. The conceptual contribution of my dissertation research is this new framework for evaluating how students think about and evaluate their programming ability and the groundwork of a new model for how this theory connects to related constructs, like self-efficacy and emotions.

Methodologically, I contribute two new research approaches for studying student perspectives of their programming experiences that combine qualitative research methods with data sources like sensors and interaction log data. The first methodological contribution is an approach for designing an automated detection systems based on student perspectives of the programming process, which I describe in Chapter 5. The approach combines AI detection systems with retrospective interviews to produce a system that can detect moments of potential negative self-assessment. The methodology produces a system that relies on student perspectives of their programming experiences instead of expert-models. The second approach uses physiological devices for gaining insights into student emotions to inform a retrospective interview, which I described in Chapter 6. Most studies during the programming process rely solely on student self-report to gain insight into their emotions, despite the challenges for students to accurately recall and express their feelings. Similarly, past studies utilizing physiological data generally rely on self-report as the gold-standard for their machine learning algorithms. My methodology uses EDA data to cue student recall during a retrospection of their programming session and improve the self-reported data. Researchers in any domain can use this methodology to improve participant accuracy for recalling emotions in a retrospective interview. My findings demonstrate the benefits of combining qualitative research methods with additional data sources like sensors or log data. Future studies should continue to use these methodologies, along with the guiding principles presented in the introduction, when designing studies that focus on student perspectives.

Finally, in this dissertation, I make a technological contribution by building a tool that can identify the moments that cause negative self-assessments from interaction log data of a programming session. We used a combination of retrospective interview data and basic artificial intelligence techniques to build this detection system. While there are many feedback systems and personalized learning environments designed for novice programmers, few systems have focused on identifying moments based on student motivation and perceptions. Most prior systems provide feedback for cognitive events, like when a student needs help, based on expert knowledge of programming process. Instead, this system identifies moments based on student perceptions of the programming process. With this system, designers of personalized learning environments for novice programmers can identify and provide feedback to students at the critical moments when they make negative self-assessments. By providing feedback at the specific moments when students may be overly critical of themselves, we can directly address students' negative perceptions of programming ability when they occur.

Across the four chapters of the dissertation, I identified three different sets of information that are important contributors to the student programming experience: self-assessment criteria, self-assessment moments, and triggers of emotions. While my studies do not provide clear explanations of the relationship between these constructs, I can note significant overlap between the lists. This overlap is demonstrated both in topic of the list item and the similarities in student quotes during interviews. To demonstrate the overlap, I provide an example quote from each of the three constructs that relate to student perspectives on planning and thinking. In the first study, we identified criteria that students use to evaluate their programming ability, one of which was *thinking and planning is not progress*. For example, one student said *"Someone is good at programming if they keep typing and don't have to sit there and think"*. Then in the second study, we found that students self-assessed throughout the programming session at moments related to the criteria. For example, related to the self-assessment criteria of *thinking and planning is not progress*, we found that students negatively self-assessed when they *spend time planning at the beginning* of a programming session. For example, one participant said to me *"Yeah I definitely feel bad when*

I have to spend time planning and can't start programming right away", demonstrating that students make negative self-assessments when they encounter a moment of spending time planning. Finally, we investigated moments that trigger students to experience emotions while programming, noticing similarities between the self-assessment moments and these triggers. Similar to spending time planning, students described that they experienced negative emotions when they did not make progress in programming, which occurred when they were not typing code in the editor. This often occurred when students spent time thinking and planning. For example, one participant said: *"Because I don't like just sitting there and just looking at the screen, not even writing anything. I like actually writing... Maybe about 30 seconds into just sitting there. I had a negative [emotion], like, 'Okay, why am I just sitting here?'"*. From this example, the similarities in the criteria, moments and events are demonstrated through the quotes, indicating a strong relationship between the lists. Future work should continue to explore the relationship.

In this dissertation, I provide a new understanding of how students experience working on programming problems. From my research, we better understand how students evaluate themselves and how students develop their perceptions of programming ability. Practically, these findings help instructors provide better support to introductory programming students. Conceptually, researchers now have a new framework with which to interpret student experiences in programming. Methodologically, researchers have a new technique for identifying critical moments from student perspectives and a new approach for capturing insights into emotion triggering events. Finally, I provide technology developers and designers the tools to design environments that intervene at the critical moments of student self-assessment to promote positive viewpoints on programming ability. Together, my work opens many opportunities for improving student perceptions of their programming intelligence and pushes computing education researchers to continue to factor student perceptions and experiences into their research agendas and tool development.

7.1 Future work

My findings open a wide range of opportunities for researchers and intervention designers to continue to further understand the student programming experience and apply these findings into new interventions. Additionally, the research agenda provides the insights necessary to begin the development of an intervention for improving student reactions when they encounter moments of frequent negative self-assessment. Specifically, in Study 2, I contribute a list of potential negative self-assessment moments. This list provides researchers with the instances of when they could intervene with a feedback message. Then, in Study 3, I designed a detection system that can automatically identify when those moments occur. This provides researchers with the awareness of when students might be making negative self-assessments. Finally, I began an exploration of how students react emotionally in those moments, which will help the intervention designers write the messaging for the feedback. Future studies should thus build on this work by designing and implementing an intervention that improves student self-efficacy. However, the limitations of my work suggests a number of other studies. For example, while we gained insights on the events that cause students to have emotional reactions, we did not have enough participants to explore if there is a relationship between student emotional reactions and the moments where they make negative self-assessments. Future work should explore this relationship to better understand how students experience the self-assessment moments. With that understanding, in combination with the findings from Study 2 on the explanations as to why students experience negative emotions at those moments, researchers should be able to design messages for the feedback system. Thus, while my research does not start the intervention design, I lay the ground work for this type of intervention to be possible.

REFERENCES

[1] *Occupational Outlook Handbook*, Sep. 2020.

[2] C. Scaffidi, M. Shaw, and B. Myers, "An Approach for Categorizing End User Programmers to Guide Software Engineering Research," p. 5, 2005.

[3] C. Scaffidi, "Workers who use spreadsheets and who program earn more than similar workers who do neither," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, NC: IEEE, Oct. 2017, pp. 233–237, ISBN: 978-1-5386-0443-4.

[4] D. Weintrop *et al.*, "Defining Computational Thinking for Mathematics and Science Classrooms," *Journal of Science Education and Technology*, vol. 25, no. 1, pp. 127–147, Feb. 2016.

[5] P. Kinnunen and B. Simon, "Experiencing programming assignments in CS1: The emotional toll," in *Proceedings of the Sixth international workshop on Computing education research*, ACM, 2010, pp. 77–86.

[6] A. Lishinski, A. Yadav, and R. Enbody, "Students' Emotional Reactions to Programming Projects in Introduction to Programming: Measurement Approach and Influence on Learning Outcomes," in *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17*, Tacoma, Washington, USA: ACM Press, 2017, pp. 30–38, ISBN: 978-1-4503-4968-0.

[7] N. Veilleux, R. Bates, C. Allendoerfer, D. Jones, J. Crawford, and T. Floyd Smith, "The relationship between belonging and ability in computer science," in *Proceeding of the 44th ACM technical symposium on Computer science education*, ACM, 2013, pp. 65–70.

[8] C. M. Lewis, R. E. Anderson, and K. Yasuhara, ""I Don't Code All Day": Fitting in Computer Science When the Stereotypes Don't Fit," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16, event-place: Melbourne, VIC, Australia, New York, NY, USA: Association for Computing Machinery, 2016, pp. 23–32, ISBN: 978-1-4503-4449-4.

[9] J. R. Shapiro and A. M. Williams, "The Role of Stereotype Threats in Undermining Girls' and Women's Performance and Interest in STEM Fields," *Sex Roles*, vol. 66, no. 3-4, pp. 175–183, Feb. 2012.

[10] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance," in *Proceedings*

of the 2016 CHI Conference on Human Factors in Computing Systems, ACM Press, 2016, pp. 1449–1461, ISBN: 978-1-4503-3362-7.

[11] A. E. Flanigan, M. S. Peteranetz, D. F. Shell, and L.-K. Soh, "Exploring Changes in Computer Science Students' Implicit Theories of Intelligence Across the Semester," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15, event-place: Omaha, Nebraska, USA, New York, NY, USA: Association for Computing Machinery, 2015, pp. 161–168, ISBN: 978-1-4503-3630-7.

[12] Q. Cutts, E. Cutts, S. Draper, P. O'Donnell, and P. Saffrey, "Manipulating mindset to positively influence introductory programming performance," in *Proceedings of the 41st ACM technical symposium on Computer science education*, ACM, 2010, pp. 431–435.

[13] J. Gorson and E. O'Rourke, "How Do Students Talk About Intelligence? An Investigation of Motivation, Self-efficacy, and Mindsets in Computer Science," in *Proceedings of the 2019 ACM Conference on International Computing Education Research - ICER '19*, Toronto ON, Canada: ACM Press, 2019, pp. 21–29, ISBN: 978-1-4503-6185-9.

[14] P. Kinnunen and B. Simon, "My program is ok – am I? Computing freshmen's experiences of doing programming assignments," *Computer Science Education*, vol. 22, no. 1, pp. 1–28, Mar. 2012.

[15] C. M. Lewis, K. Yasuhara, and R. E. Anderson, "Deciding to major in computer science: A grounded theory of students' self-assessment of ability," in *Proceedings of the seventh international workshop on Computing education research*, ACM, 2011, pp. 3–10.

[16] A. Bandura, *Self-efficacy: The exercise of control*. Macmillan, 1997.

[17] ——, "Self-efficacy," *The Corsini encyclopedia of psychology*, pp. 1–3, 2010, Publisher: Wiley Online Library.

[18] D. H. Schunk, "Self-efficacy, motivation, and performance," *Journal of Applied Sport Psychology*, vol. 7, no. 2, pp. 112–137, Sep. 1995.

[19] J. D. Relich, R. L. Debus, and R. Walker, "The mediating role of attribution and self-efficacy variables for treatment effects on achievement outcomes," *Contemporary Educational Psychology*, vol. 11, no. 3, pp. 195–216, 1986, Publisher: Elsevier.

[20] C. Watson, F. W. Li, and J. L. Godwin, "No tests required: Comparing traditional and dynamic predictors of programming success," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 469–474.

[21] A. Lishinski, A. Yadav, J. Good, and R. Enbody, "Learning to Program: Gender Differences and Interactive Effects of Students' Motivation, Goals, and Self-Efficacy on Performance,"

in *In Proceedings of the 2016 ACM Conference on International Computing Education Research*, ACM Press, 2016, pp. 211–220, ISBN: 978-1-4503-4449-4.

[22] V. Ramalingam, D. LaBelle, and S. Wiedenbeck, "Self-efficacy and mental models in learning to program," in *ACM SIGCSE Bulletin*, vol. 36, ACM, 2004, pp. 171–175.

[23] A. Bandura, "Self-efficacy mechanism in human agency.," *American psychologist*, vol. 37, no. 2, p. 122, 1982.

[24] P. Kinnunen and B. Simon, "CS majors' self-efficacy perceptions in CS1: Results in light of social cognitive theory," in *Proceedings of the seventh international workshop on Computing education research*, ACM, 2011, pp. 19–26.

[25] A. V. Robins, "Novice programmers and introductory programming," *The Cambridge handbook of computing education research*, vol. 1, pp. 327–376, 2019.

[26] C. M. Lawson, "A survey of computer facility management," *Datamotion*, vol. 8, no. 7, pp. 29–32, 1962.

[27] R. D. Pea and D. M. Kurland, "On the Cognitive Prerequisites of Learning Computer Programming," p. 93, 1983.

[28] J. S. Olson and W. A. Kellogg, *Ways of Knowing in HCI*. Springer, 2014, vol. 2.

[29] B. Parkinson and A. S. R. Manstead, "Making sense of emotion in stories and social life," *Cognition & Emotion*, vol. 7, no. 3-4, pp. 295–323, May 1993.

[30] J. Gorson and E. O'Rourke, "CS1 Student Assessments of Themselves Relative to Others: The Role of Self-Critical Bias and Gender," in *Proceedings of the 15th International Conference of the Learning Sciences-ICLS 2021.*, International Society of the Learning Sciences, 2021.

[31] J. Gorson and E. O'Rourke, "Why do CS1 Students Think They're Bad at Programming? Investigating Self-efficacy and Self-assessments at Three Universities," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, 2020, pp. 170–181.

[32] J. Gorson, N. LaGrassa, C. H. Hu, E. Lee, A. M. Robinson, and E. O'Rourke, "An Approach for Detecting Student Perceptions of the Programming Experience from Interaction Log Data," in *Artificial Intelligence in Education*, Series Title: Lecture Notes in Computer Science, vol. 12748, Cham: Springer International Publishing, 2021, pp. 150–164, ISBN: 978-3-030-78291-7 978-3-030-78292-4.

[33] J. Gorson, K. Cunningham, M. Worsley, and E. O'Rourke, "Using electrodermal activity measurements to understand novice programmer emotions (In Review)," in *Proceedings of*

the 2022 ACM Conference on International Computing Education Research - ICER '22, 2022.

[34] C. Watson and F. W. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, Uppsala, Sweden: ACM Press, 2014, pp. 39–44, ISBN: 978-1-4503-2833-3.

[35] L. Ott, B. Bettin, and L. Ureel, "The impact of placement in introductory computer science courses on student persistence in a computing major," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 2018, pp. 296–301.

[36] C. M. Mueller and C. S. Dweck, "Praise for intelligence can undermine children's motivation and performance.," *Journal of personality and social psychology*, vol. 75, no. 1, p. 33, 1998.

[37] S. R. Zentall and B. J. Morris, ""Good job, you're so smart": The effects of inconsistency of praise type on young children's motivation," *Journal of Experimental Child Psychology*, vol. 107, no. 2, pp. 155–163, Oct. 2010.

[38] A. Master, S. Cheryan, and A. N. Meltzoff, "Computing whether she belongs: Stereotypes undermine girls' interest and sense of belonging in computer science.," *Journal of educational psychology*, vol. 108, no. 3, p. 424, 2016, Publisher: American Psychological Association.

[39] A. Fisher and J. Margolis, "Unlocking the clubhouse: The Carnegie Mellon experience," *ACM SIGCSE Bulletin*, vol. 34, no. 2, pp. 79–83, 2002.

[40] J. M. Cohoon, "Just get over it or just get on with it," *Retaining women in undergraduate computing. In J. Cohoon & W. Aspray (Eds.), Women and information technology: Research on underrepresentation*, pp. 205–238, 2006.

[41] C. S. Dweck, *Self-theories: Their role in motivation, personality, and development*. Psychology Press, 1999, ISBN: 1-84169-024-4.

[42] C. S. Dweck and E. L. Leggett, "A social-cognitive approach to motivation and personality.," *Psychological review*, vol. 95, no. 2, p. 256, 1988.

[43] C. S. Dweck, *Mindset: The new psychology of success*. Random House Incorporated, 2006.

[44] L. S. Blackwell, K. H. Trzesniewski, and C. S. Dweck, "Implicit theories of intelligence predict achievement across an adolescent transition: A longitudinal study and an intervention," *Child development*, vol. 78, no. 1, pp. 246–263, 2007.

[45] C. Good, J. Aronson, and M. Inzlicht, "Improving adolescents' standardized test performance: An intervention to reduce the effects of stereotype threat," *Journal of Applied Developmental Psychology*, vol. 24, no. 6, pp. 645–662, Dec. 2003.

[46] J. Aronson, C. B. Fried, and C. Good, "Reducing the Effects of Stereotype Threat on African American College Students by Shaping Theories of Intelligence," *Journal of Experimental Social Psychology*, vol. 38, no. 2, pp. 113–125, Mar. 2002.

[47] D. S. Yeager *et al.*, "Using design thinking to improve psychological interventions: The case of the growth mindset during the transition to high school.," *Journal of Educational Psychology*, vol. 108, no. 3, pp. 374–391, 2016.

[48] E. O'Rourke, K. Haimovitz, C. Ballweber, C. Dweck, and Z. Popović, "Brain points: A growth mindset incentive structure boosts persistence in an educational game," in *Proceedings of the SIGCHI conference on human factors in computing systems*, ACM, 2014, pp. 3339–3348.

[49] A. T. Corbett and J. R. Anderson, "Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 2001, pp. 245–252.

[50] B. Simon *et al.*, "Saying isn't necessarily believing: Influencing self-theories in computing," in *Proceedings of the Fourth international Workshop on Computing Education Research*, ACM, 2008, pp. 173–184.

[51] D. H. Schunk, "Attributions and Perceptions of Efficacy during Self-Regulated Learning by Remedial Readers.," 1989.

[52] N. E. Betz and G. Hackett, "The relationship of mathematics self-efficacy expectations to the selection of science-based college majors," *Journal of Vocational Behavior*, vol. 23, no. 3, pp. 329–345, Dec. 1983.

[53] G. Hackett and N. E. Betz, "A self-efficacy approach to the career development of women," *Journal of Vocational Behavior*, vol. 18, no. 3, pp. 326–339, Jun. 1981.

[54] R. W. Lent and G. Hackett, "Career self-efficacy: Empirical status and future directions," *Journal of Vocational Behavior*, vol. 30, no. 3, pp. 347–382, Jun. 1987.

[55] F. Pajares and M. D. Miller, "Role of self-efficacy and self-concept beliefs in mathematical problem solving: A path analysis.," *Journal of educational psychology*, vol. 86, no. 2, p. 193, 1994, Publisher: American Psychological Association.

[56] I. T. Miura, "The relationship of computer self-efficacy expectations to computer interest and course enrollment in college," *Sex Roles*, vol. 16, no. 5-6, pp. 303–311, Mar. 1987.

[57] F. B. Tek, K. S. Benli, and E. Deveci, “Implicit Theories and Self-Efficacy in an Introductory Programming Course,” *IEEE Transactions on Education*, vol. 61, no. 3, pp. 218–225, Aug. 2018.

[58] J. Hui, M. Greenberg, and E. Gerber, “Understanding Crowdfunding Work: Implications for Support Tools,” in *CHI ’13 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA ’13, event-place: Paris, France, New York, NY, USA: Association for Computing Machinery, 2013, pp. 889–894, ISBN: 978-1-4503-1952-2.

[59] E. Harburg, J. Hui, M. Greenberg, and E. M. Gerber, “Understanding the Effects of Crowdfunding on Entrepreneurial Self-Efficacy,” in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW ’15, event-place: Vancouver, BC, Canada, New York, NY, USA: Association for Computing Machinery, 2015, pp. 3–16, ISBN: 978-1-4503-2922-4.

[60] P. Shea and T. Bidjerano, “Learning presence: Towards a theory of self-efficacy, self-regulation, and the development of a communities of inquiry in online and blended learning environments,” *Computers & education*, vol. 55, no. 4, pp. 1721–1731, 2010, Publisher: Elsevier.

[61] S. Marwan, G. Gao, S. Fisk, T. W. Price, and T. Barnes, “Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science,” in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, Virtual Event New Zealand: ACM, Aug. 2020, pp. 194–203, ISBN: 978-1-4503-7092-9.

[62] B. Hasan, “The influence of specific computer experiences on computer self-efficacy beliefs,” *Computers in Human Behavior*, vol. 19, no. 4, pp. 443–450, Jul. 2003.

[63] P. Askar and D. Davenport, “An investigation of factors related to self-efficacy for Java programming among engineering students,” *TOJET: The Turkish Online Journal of Educational Technology*, vol. 8, no. 1, 2009.

[64] D. W. Govender and S. K. Basak, “An investigation of factors related to self-efficacy for Java programming among computer science education students,” *Journal of Governance and Regulation*, vol. 4, no. 4, pp. 612–619, 2015.

[65] S. A. Ambrose, Ed., *How learning works: seven research-based principles for smart teaching*, 1st ed, ser. The Jossey-Bass higher and adult education series. San Francisco, CA: Jossey-Bass, 2010, OCLC: ocn468969206, ISBN: 978-0-470-48410-4.

[66] D. L. Butler and P. H. Winne, “Feedback and self-regulated learning: A theoretical synthesis,” *Review of educational research*, vol. 65, no. 3, pp. 245–281, 1995.

[67] C. Quintana, M. Zhang, and J. Krajcik, “A Framework for Supporting Metacognitive Aspects of Online Inquiry Through Software-Based Scaffolding,” *Educational Psychologist*, vol. 40, no. 4, pp. 235–244, Dec. 2005.

[68] M. D. Alicke, “Global self-evaluation as determined by the desirability and controllability of trait adjectives.,” *Journal of Personality and Social Psychology*, vol. 49, no. 6, pp. 1621–1630, 1985.

[69] J. D. Brown, “Evaluations of Self and Others: Self-Enhancement Biases in Social Judgments,” *Social Cognition*, vol. 4, no. 4, pp. 353–376, Dec. 1986.

[70] V. S. Y. Kwan, O. P. John, D. A. Kenny, M. H. Bond, and R. W. Robins, “Reconceptualizing Individual Differences in Self-Enhancement Bias: An Interpersonal Approach.,” *Psychological Review*, vol. 111, no. 1, pp. 94–110, 2004.

[71] J. Kruger and D. Dunning, “Unskilled and unaware of it: How difficulties in recognizing one’s own incompetence lead to inflated self-assessments.,” *Journal of personality and social psychology*, vol. 77, no. 6, p. 1121, 1999, Publisher: American Psychological Association.

[72] K. W. Eva and G. Regehr, ““I’ll never play professional football” and other fallacies of self-assessment,” *Journal of Continuing Education in the Health Professions*, vol. 28, no. 1, pp. 14–19, 2008.

[73] J. Ehrlinger and D. Dunning, “How chronic self-views influence (and potentially mislead) estimates of performance.,” *Journal of personality and social psychology*, vol. 84, no. 1, p. 5, 2003, Publisher: American Psychological Association.

[74] H. Ginsburg, *Entering the child’s mind: The clinical interview in psychological research and practice*. New York: Cambridge University Press, 1997.

[75] M. J. Scott and G. Ghinea, “On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice,” *IEEE Transactions on Education*, vol. 57, no. 3, pp. 169–174, 2014.

[76] M. B. Miles, A. M. Huberman, and J. Saldana, *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[77] D. R. Thomas, “A general inductive approach for analyzing qualitative evaluation data,” *American journal of evaluation*, vol. 27, no. 2, pp. 237–246, 2006.

[78] A. Strauss and J. Corbin, “Grounded theory methodology,” *Handbook of qualitative research*, vol. 17, pp. 273–85, 1994.

[79] C. S. Dweck and J. Bempechat, "Children's theories of intelligence: Consequences for learning," *Learning and motivation in the classroom*, pp. 239–256, 1983.

[80] J. S. Eccles and B. L. Barber, "Student council, volunteering, basketball, or marching band what kind of extracurricular involvement matters?" *Journal of adolescent research*, vol. 14, no. 1, pp. 10–43, 1999.

[81] T. Byrt, J. Bishop, and J. B. Carlin, "Bias, prevalence and kappa," *Journal of Clinical Epidemiology*, vol. 46, no. 5, pp. 423–429, May 1993.

[82] H. De Vries, M. N. Elliott, D. E. Kanouse, and S. S. Teleki, "Using Pooled Kappa to Summarize Interrater Agreement across Many Items," *Field Methods*, vol. 20, no. 3, pp. 272–282, Aug. 2008.

[83] K. A. Hallgren, "Computing inter-rater reliability for observational data: An overview and tutorial," *Tutorials in quantitative methods for psychology*, vol. 8, no. 1, p. 23, 2012.

[84] K. A. Ericsson and H. A. Simon, *Protocol analysis: Verbal reports as Data*. Cambridge, MA: MIT Press, 1984.

[85] A. Ebrahimi, "Novice programmer errors: Language constructs and plan composition," *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994.

[86] R. McCartney, A. Eckerdal, J. E. Mostrom, K. Sanders, and C. Zander, "Successful students' strategies for getting unstuck," in *ACM SIGCSE Bulletin*, vol. 39, ACM, 2007, pp. 156–160, ISBN: 1-59593-610-6.

[87] B. Hanks and M. Brandt, "Successful and unsuccessful problem solving approaches of novice programmers," in *ACM SIGCSE Bulletin*, vol. 41, ACM, 2009, pp. 24–28.

[88] S. Sonnentag, "Expertise in professional software design: A process study.," *Journal of applied psychology*, vol. 83, no. 5, p. 703, 1998, Publisher: American Psychological Association.

[89] W. Visser, "Strategies in programming programmable controllers: A field study on a professional programmer," in *Empirical Studies of Programmers: Second workshop (ESP2)*, Ablex, 1987, pp. 217–230.

[90] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: The good, the bad, and the quirky–a qualitative analysis of novices' strategies," in *ACM SIGCSE Bulletin*, vol. 40, ACM, 2008, pp. 163–167.

[91] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceeding of the 28th international conference on Software engineering - ICSE '06*, Shanghai, China: ACM Press, 2006, p. 492, ISBN: 978-1-59593-375-1.

[92] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, “Studying the advancement in debugging practice of professional software developers,” *Software Quality Journal*, vol. 25, no. 1, pp. 83–110, Mar. 2017.

[93] J. Kurman, “Gender, Self-Enhancement, and Self-Regulation of Learning Behaviors in Junior High School,” *Sex Roles*, vol. 50, no. 9/10, pp. 725–735, May 2004.

[94] C. Midgley *et al.*, “Patterns of Adaptive Learning Scales,” American Psychological Association, Tech. Rep., Apr. 2013, type: dataset.

[95] J. Cohen D, L. Margulieux, M. Renken, and W. M. Jones, “Conclusions from the Validation of a Vignette-Based Instrument to Measure Maker MIndsets,” in *Proceedings of the Fifteenth International Conference for the Learning Sciences (ICLS) 2020*, Nashville, TN, USA: International Society of the Learning Sciences, Inc.[ISLS]., Jun. 2020.

[96] C. S. Alexander and H. J. Becker, “The Use of Vignettes in Survey Research,” *Public Opinion Quarterly*, vol. 42, no. 1, p. 93, 1978.

[97] S. E. Carlson, D. G. Rees Lewis, E. M. Gerber, and M. W. Easterday, “Challenges of peer instruction in an undergraduate student-led learning community: Bi-directional diffusion as a crucial instructional process,” *Instructional Science*, vol. 46, no. 3, pp. 405–433, Jun. 2018.

[98] J. L. Collett and E. Childs, “Minding the gap: Meaning, affect, and the potential shortcomings of vignettes,” *Social Science Research*, vol. 40, no. 2, pp. 513–522, Mar. 2011.

[99] A. W. Bowman and A. Azzalini, *Applied smoothing techniques for data analysis: the kernel approach with S-Plus illustrations*. OUP Oxford, 1997, vol. 18.

[100] J. Bennedsen and M. E. Caspersen, “Revealing the programming process,” in *ACM SIGCSE Bulletin*, vol. 37, ACM, 2005, pp. 186–190, ISBN: 1-58113-997-7.

[101] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, “Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming,” *Journal of the Learning Sciences*, vol. 23, no. 4, pp. 561–599, 2014.

[102] P. Blikstein, “Using learning analytics to assess students’ behavior in open-ended programming tasks,” in *Proceedings of the 1st international conference on learning analytics and knowledge*, ACM, 2011, pp. 110–116.

[103] M. Berland, T. Martin, T. Benton, C. Petrick Smith, and D. Davis, “Using learning analytics to understand the learning pathways of novice programmers,” *Journal of the Learning Sciences*, vol. 22, no. 4, pp. 564–599, 2013.

[104] M. C. Jadud, “Methods and Tools for Exploring Novice Compilation Behaviour,” in *Proceedings of the Second International Workshop on Computing Education Research*, ser. ICER ’06, event-place: Canterbury, United Kingdom, New York, NY, USA: Association for Computing Machinery, 2006, pp. 73–84, ISBN: 1-59593-494-4.

[105] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen, “Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance,” in *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER ’15*, Omaha, Nebraska, USA: ACM Press, 2015, pp. 121–130, ISBN: 978-1-4503-3630-7.

[106] J. P. Munson and J. P. Zitovsky, “Models for early identification of struggling novice programmers,” in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 699–704.

[107] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein, “Modeling how students learn to program,” in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, ACM, 2012, pp. 153–160.

[108] S. Edwards and Z. Li, “Towards progress indicators for measuring student programming effort during solution development,” in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling ’16*, Koli, Finland: ACM Press, 2016, pp. 31–40, ISBN: 978-1-4503-4770-9.

[109] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, “Cognitive tutors: Lessons learned,” *The journal of the learning sciences*, vol. 4, no. 2, pp. 167–207, 1995.

[110] J. R. Anderson, F. G. Conrad, and A. T. Corbett, “Skill acquisition and the LISP tutor,” *Cognitive Science*, vol. 13, no. 4, pp. 467–505, 1989, Publisher: Elsevier.

[111] A. Corbett, “Cognitive computer tutors: Solving the two-sigma problem,” in *International Conference on User Modeling*, Springer, 2001, pp. 137–147.

[112] E. Soloway, J. Bonar, and K. Ehrlich, “Cognitive strategies and looping constructs: An empirical study,” *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, Nov. 1983.

[113] M. Fuchs, M. Heckner, F. Raab, and C. Wolff, “Monitoring students’ mobile app coding behavior data analysis based on IDE and browser interaction logs,” in *2014 IEEE Global Engineering Education Conference (EDUCON)*, Istanbul: IEEE, Apr. 2014, pp. 892–899, ISBN: 978-1-4799-3191-0.

[114] B. J. Reiser, J. R. Anderson, and R. G. Farrell, “Dynamic Student Modelling in an Intelligent Tutor for LISP Programming.,” in *IJCAI*, vol. 85, 1985, pp. 8–14.

[115] J. H. Cross, D. Hendrix, and D. A. Umphress, "JGRASP: An integrated development environment with visualizations for teaching java in CS1, CS2, and beyond," in *34th Annual Frontiers in Education, 2004. FIE 2004.*, IEEE Computer Society, 2004, pp. 1466–1467.

[116] A. K. Shenton, "Strategies for ensuring trustworthiness in qualitative research projects," *Education for Information*, vol. 22, no. 2, pp. 63–75, Jul. 2004.

[117] B. Saunders *et al.*, "Saturation in qualitative research: Exploring its conceptualization and operationalization," *Quality & Quantity*, vol. 52, no. 4, pp. 1893–1907, Jul. 2018.

[118] M. O'Reilly and N. Parker, "'Unsatisfactory Saturation': A critical exploration of the notion of saturated sample sizes in qualitative research," *Qualitative Research*, vol. 13, no. 2, pp. 190–197, Apr. 2013.

[119] L. A. Zadeh, "Fuzzy logic," *Computer*, vol. 21, no. 4, pp. 83–93, 1988, Publisher: IEEE.

[120] M. Coto, S. Mora, B. Grass, and J. Murillo-Morera, "Emotions and programming learning: Systematic mapping," *Computer Science Education*, pp. 1–36, May 2021.

[121] J. Chetty, ""I hate programming" and Other Oscillating Emotions Experienced by Novice Students Learning Computer Programming," p. 7, 2013.

[122] N. Bosch, S. D'Mello, and C. Mills, "What Emotions Do Novices Experience during Their First Computer Programming Learning Session?" In *Artificial Intelligence in Education*, H. C. Lane, K. Yacef, J. Mostow, and P. Pavlik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 11–20, ISBN: 978-3-642-39112-5.

[123] N. Bosch and S. D'Mello, "The Affective Experience of Novice Computer Programmers," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 181–206, Mar. 2017.

[124] P. Haden, D. Parsons, K. Wood, and J. Gasson, "Student affect in CS1: Insights from an easy data collection tool," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Nov. 2017, pp. 40–49, ISBN: 978-1-4503-5301-4.

[125] D. Graziotin, X. Wang, and P. Abrahamsson, "Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering," p. 21, 2014.

[126] D. Girardi, N. Novielli, D. Fucci, and F. Lanubile, "Recognizing developers' emotions while programming," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Seoul South Korea: ACM, Jun. 2020, pp. 666–677, ISBN: 978-1-4503-7121-6.

[127] S. C. Muller and T. Fritz, “Stuck and Frustrated or in Flow and Happy: Sensing Developers’ Emotions and Progress,” p. 12, 2015.

[128] I. Drosos, P. J. Guo, and C. Parnin, “HappyFace: Identifying and predicting frustrating obstacles for learning programming at scale,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*, IEEE, 2017, pp. 171–179.

[129] J. J. Braithwaite, D. G. Watson, R. Jones, and M. Rowe, “A guide for analysing electrodermal activity (EDA) & skin conductance responses (SCRs) for psychological experiments,” *Psychophysiology*, vol. 49, no. 1, pp. 1017–1034, 2013.

[130] P. Blikstein and M. Worsley, “Multimodal Learning Analytics and Education Data Mining: Using Computational Technologies to Measure Complex Learning Tasks,” *Journal of Learning Analytics*, vol. 3, no. 2, pp. 220–238, Sep. 2016.

[131] W. Boucsein, *Electrodermal activity*. Springer Science & Business Media, 2012.

[132] T. Bentley, L. Johnston, and K. von Baggo, “Evaluation using cued-recall debrief to elicit information about a user’s affective experiences,” in *Proceedings of the 17th Australia Conference on Computer-Human Interaction: Citizens Online: Considerations for Today and the Future*, 2005, pp. 1–10.

[133] M. E. Dawson, A. M. Schell, and D. L. Filion, “The electrodermal system,” 2017, Publisher: Cambridge University Press.

[134] K. R. Scherer, “What are emotions? And how can they be measured?” *Social Science Information*, vol. 44, no. 4, pp. 695–729, Dec. 2005.

[135] *APA dictionary of psychology*. American Psychological Association.

[136] M. M. T. Rodrigo and R. S. Baker, “Coarse-grained detection of student frustration in an introductory programming course,” in *Proceedings of the fifth international workshop on Computing education research workshop - ICER ’09*, Berkeley, CA, USA: ACM Press, 2009, p. 75, ISBN: 978-1-60558-615-1.

[137] I. Drosos, P. J. Guo, and C. Parnin, “HappyFace: Identifying and predicting frustrating obstacles for learning programming at scale,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*, IEEE, 2017, pp. 171–179.

[138] H. D. Critchley, J. Eccles, and S. N. Garfinkel, “Interaction between cognition, emotion, and the autonomic nervous system,” in *Handbook of clinical neurology*, vol. 117, Elsevier, 2013, pp. 59–77.

[139] D. Caruelle, A. Gustafsson, P. Shams, and L. Lervik-Olsen, “The use of electrodermal activity (EDA) measurement to understand consumer emotions – A literature review and a call for action,” *Journal of Business Research*, vol. 104, pp. 146–160, Nov. 2019.

[140] *E4 wristband | Real-time physiological signals | Wearable PPG, EDA, Temperature, Motion sensors*.

[141] D. Lunn and S. Harper, “Using Galvanic Skin Response Measures to Identify Areas of Frustration for Older Web 2.0 Users,” in *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, ser. W4A ’10, event-place: Raleigh, North Carolina, New York, NY, USA: Association for Computing Machinery, 2010, ISBN: 978-1-4503-0045-2.

[142] M.-H. Choi *et al.*, “Changes in Cognitive Performance Due to Three Types of Emotional Tension,” in *Database Theory and Application, Bio-Science and Bio-Technology*, Y. Zhang, A. Cuzzocrea, J. Ma, K.-i. Chung, T. Arslan, and X. Song, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–264, ISBN: 978-3-642-17622-7.

[143] M. Worsley and P. Blikstein, “A multimodal analysis of making,” *International Journal of Artificial Intelligence in Education*, vol. 28, no. 3, pp. 385–419, 2018, Publisher: Springer.

[144] Y. S. Can, N. Chalabianloo, D. Ekiz, and C. Ersoy, “Continuous Stress Detection Using Wearable Sensors in Real Life: Algorithmic Programming Contest Case Study,” *Sensors*, vol. 19, no. 8, p. 1849, Apr. 2019.

[145] L. Ahonen, B. U. Cowley, A. Hellas, and K. Puolamäki, “Biosignals reflect pair-dynamics in collaborative work: EDA and ECG study of pair-programming in a classroom environment,” *Scientific Reports*, vol. 8, no. 1, p. 3138, Dec. 2018.

[146] M. Wrobel, “Applicability of Emotion Recognition and Induction Methods to Study the Behavior of Programmers,” *Applied Sciences*, vol. 8, no. 3, p. 323, Feb. 2018.

[147] K. Nolan, A. Mooney, and S. Bergin, “An Investigation of Gender Differences in Computer Science Using Physiological, Psychological and Behavioural Metrics,” in *Proceedings of the Twenty-First Australasian Computing Education Conference on - ACE ’19*, Sydney, NSW, Australia: ACM Press, 2019, pp. 47–55, ISBN: 978-1-4503-6622-9.

[148] C. McCarthy, N. Pradhan, C. Redpath, and A. Adler, “Validation of the Empatica E4 wristband,” in *2016 IEEE EMBS International Student Conference (ISC)*, Ottawa, ON, Canada: IEEE, May 2016, pp. 1–4, ISBN: 978-1-5090-0935-0.

[149] A. Bruun, E. L.-C. Law, T. D. Nielsen, and M. Heintz, “Do You Feel the Same? On the Robustness of Cued-Recall Debriefing for User Experience Evaluation,” *ACM Transactions on Computer-Human Interaction*, vol. 28, no. 4, pp. 1–45, Oct. 2021.

[150] M. T. Orne, “On the social psychology of the psychological experiment: With particular reference to demand characteristics and their implications.,” *American Psychologist*, vol. 17, no. 11, 1962.

[151] S. Taylor, N. Jaques, Weixuan Chen, S. Fedor, A. Sano, and R. Picard, “Automatic identification of artifacts in electrodermal activity data,” in *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Milan: IEEE, Aug. 2015, pp. 1934–1937, ISBN: 978-1-4244-9271-8.

[152] J. Corbin and A. Strauss, *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, California, May 2020.

[153] J. Saldaña, *The coding manual for qualitative researchers*. sage, 2021.

APPENDIX A
QUALITATIVE CODEBOOK FOR RESEARCHERS TO IDENTIFY MOMENTS OF POTENTIAL SELF-ASSESSMENT

This codebook was designed during Study 3, discussed in Chapter 5. We used an iterative qualitative procedure to design and validate the codebook. The codebook is designed for researchers to use while watching secreen recordings of a programming session to identify the moments that align with the self-assessment moments, whether or not the student self-assess. In addition, the codebook informed the design of the automated tool for detecting the moments.

A.1 Guide for labeling using resources moments

Using resources labels are used for the moments when participants use resources to get help on solving the programming problem. While some participants consider internal IDE features, such as the debugger, console messages, or a new, empty file, to be resources; in this codebook, we only consider using resources labels for when they use external resources, such as using the browser or opening an existing file.

Using resources moments are further divided into three labels: using resources for syntax, using resources for approach, and using resources in general. We are interested in identifying when students use resources for approach or syntax, however, since there is some overlap in the way participants interpret approach versus syntax, as two participants might label the same behavior as different moments, we use the using resources in general label for behaviors that do not clearly fit either one or the other category. Specifically, if a participant's behaviors are not particularly indicative of either syntax or approach, label that moments the using resources in general label. On the other hand, if we think that a participant is using resources for both syntax and approach in the session, we will label it approach. This is because using resources for syntax should only label moments where a participant purely uses the resources for syntax and nothing else, as it

is likely that if a participant researches an approach, they might end up learning about syntax simultaneously.

When deciding which of the three sub-labels to use, the participant's purpose of using resources should be the main factor to consider. However, since we can not always know the purpose while watching a participant programming, we present five different types of indicators for determining what students used the resources for: 1) behaviors before using resource, 2) time on resource, 3) search terms (if applicable), 4) behaviors while using resources, 5) behaviors when returning. To make a decision, you should evaluate across the groups to see if there is more indication in one direction compared to others. The decision needs to be a conglomerate and not just based on time, even if time is a strong indicator. When there is conflict, the search terms should get priority if they are clear.

A.1.1 Definitions of terms and concepts

Below are the terms and concepts that are applicable to all three using resources labels.

Types of resources

- Search results are results from web search engines such as Google or Bing
- Online course materials are material provided by a course, generally accessed course websites
- Existing files on the participant's computer or personal cloud folder that they open with jGRASP - generally past assignments or practice problems or examples from their current or previous class
- Other files that are not Java files and are either on the participant's local computer or in their personal cloud folder, for example, a PDF document or course slide that is saved locally on their computer

Broad search term

Search terms that only contain the name of a data type plus “java” or “java method” without specifying a method, for example “string java”, “arraylist java method”, “scanner java”.

Indistinct search term

When the participant conducts more than one search in the same using resources session, if the new search term fits the following situations, the new search term will be considered as indistinct:

- Adds “java” to the beginning or the end of the search term and search again (generally because they did not add “java” to the search term and thus get search results in other programming languages).
- There is a typo in the previous search term, so participant either click the suggested search term given by the search engine or manually re-enter the correct search term
- Rearranges words in the previous search term and/or adding extra word(s) that does not affect the meaning of the search term. For example: “string java method” to “java method for string”, “arraylist java remove” to “arraylist java 8 remove”

Online documentation of a Java class

Documentation is defined as any formal write-up of a class and usually contains a list of methods supported by the class, such as full documentation of Class String. It is not a forum, example or blogpost like Stack Overflow. Oracle documentation is the mostly commonly used website for documentation by participants. We have also seen participants using tutorials point for such documentation.

Moments that do not qualify for any of the using resources labels

- Spent less than 3 seconds on the resources (or such a short period of time that they clearly could not have learned something from the experience)

- The problem description is the only site they visit
- Did a search but did not click into any search results after the results page is loaded and leave shortly after they search without using the resource. Likely if they quickly scrolled and skimmed through the search results, they realized that this is not helpful, so they either do another search/ use a different type of resource/ go back to the IDE instead. Be aware: this is not to be mistaken for when participants use the preview of search results as their source of information and thus do not need to click into any site. This would count as a label, see using resource for syntax description for more details.
- Looked at the table of contents or index of a resource (often a class syllabus) but does not click into any particular reference page.

When to count a new session of using resources

When a participant uses a resource, the following situations will be considered as a new session for using resources and do not count as toggling:

- Go to a new site not linked on current site. For example, participant enters a url or a bookmark in the browser. If they click to a link from the current page (not the search result page) that leads to a new site, it will not be a new session
- Go to a different search result
- Start a new search
- Open a new existing file
- Switch to a different file or a different tab in the browser than the file/ tab that they were last on in the most previous using resources session (assuming the different file/ tab is already opened either in the IDE or the browser)

It is common to see participants toggling between the IDE and the resource while they are implementing the syntax of a method or syntax of an approach. Toggling should not be a new label.

Toggling must include these factors:

- Return to the resource within a short pause or they are still implementing the information learned from the resources (for reference: around 1 minute since they left). If in the time when they are back in the IDE qualifies for stopping to think, it should be a new label for using resources
- The resource they go back to must be the same one as the most recent using resources label
- The resource must already be open on the computer; meaning they did not close the tab/file the last time they left

If they flip to the problem description or go back the IDE for a very short period of time (like a quick flip to look at something quickly or while changing views; approximately less than 5 seconds) in the middle of using resources, it will be considered as the same using resources session. If they pause on the problem description or the IDE for longer or make changes in the editor, see rules for toggling.

When to start/stop measuring time spent on a certain type of resource

- If the participant is using google search, the timing starts when the browser fully loads the search result
- If the participant is using online course resource, the timing starts after they navigate through the table of contents (or anything similar to that, if applicable) on the course website and click into a page/lesson/file etc. Could get to the course website through search engine or bookmark on browser or a link they kept somewhere else
- If the participant is using an existing file within jGRASP OR a local file on their computer (for example a PDF document or slides) OR a file on their personal cloud folder, the timing starts when they open the file

- In all cases, whenever focus returns to jGRASP (or if they are using a file as a resource, the focus returns to the main file) OR go to the problem description while they are on the browser, the resource timer stops.

- In all cases, if there are multiple instances of using resources, such as using more than one type of resources, opening multiple files within jGRASP, and clicking into different search results or doing another search, the total time on resources is accumulated across different instances as long as they are all in the same session. The total time will be used for evaluating time indicator, unless more detailed situations are specified.

Situations with multiple instances of using resources in one session

In situations where there are multiple instances of using resources, especially when they do multiple distinct searches in the same session, only if all instances are clear to be for syntax individually would we label the session as using resources for syntax. If there are both approach and syntax related searches, then the moment should be labelled approach.

A.1.2 Using resource for syntax

See participant leave jGRASP IDE or open a different file within jGRASP.

Purposes for using resources that count as syntax

Should only label moments as using resources for syntax where a participant purely uses the resources for syntax and nothing else.

- Syntax of [specific method calls]: this is under the condition that the participant has an idea of what method to use, and they are unsure of syntax of that method

- Syntax for [declaring a non-primitive data type of variable], for example: int array, arraylist, hashset

- Syntax for [importing packages lines]

- Syntax for [loops or conditional statements], such as syntax for a for-each loop
- Look up how to implement similar functions or methods from other programming languages
- Coding style guide (most likely from course material)

Behavior before going to resources

Positive indicators:

Strong:

- Stopped typing in the middle of implementing a method or just finished typing a method and immediately goes to a resource to look up that method. In this case, the method they just typed should be relevant to the content of the resource. "In the middle of implementing a method" refers to situations when the participant already typed out part of the method name (for example, ".toLower" if intended to do ".toLowerCase()") or typed the full method name but not yet added the required parameters
- Gets simple java error and goes directly to a resource (does not get label stopping to think before deciding to use resource). For example: getting "error: cannot find symbol" and realizing that they forget to import package(s), and thus using resources for syntax of the importing lines

Middle:

- Stopped typing in the middle of a line in general
- Could be in the middle of setting up a loop, a conditional statement, etc., but not in the middle of implementing a method

Negative indications:

- Stopping to think before using the resource
- Occurring before implementation starts

Behaviors while using resources

Time spent on resources for syntax

Most of the participants spend less than 1 minute on resources if they are using resources for syntax (if the time is under 30 seconds then it is even more likely using resources). However, some participants could be working on the problem at either a faster or slower pace than other participants. Therefore, search terms and other behaviors should be prioritized. when total time exceeds 30 seconds.

If the situation fully fits the description for special cases, we would use special cases for time on resources instead of total time for evaluating time indication. If not, use total time.

Total time

Strong indication:

- 3 - 20 seconds, very likely to be using resources for syntax even without seeing other behaviors

Middle indication:

- 20 - 30 seconds

Neutral:

- 30 seconds - 1 minute

Negative indication:

- 1 minute and 30 seconds, especially the participant looks at more than one previous files within jGRASP or conducts more than one distinct search without leaving the browser

Special cases for time on resources

Spent a total of less than 1 minute on two search results without leaving the browser (under the same search term or indistinct search terms)

When participant leaves jGRASP IDE entirely

To decide if a chrome session was using resources for syntax, look for the following things:

Strong:

- Search terms (when using a search engine) that serve as strong indicator for syntax:
- Search term that contains a specific java method
- Example search terms: “string length java”, “substring java”, “tostring java”, “arraylist java remove”
- Search term that highly resembles the purpose of a specific java method
- Example search terms: “Java how to find length of string”, “Make all values in string lower-case java”, “Splitting things into individual words java”
- Specifying about how to declare a variable with non-primitive data type such as arraylist, int array
- Example search terms: “Declare int array java”, “creating an arraylist in java”
- Using Google search and then clicking into image search results for example syntax instead of website results;
- Using broad search term and navigating to online documentation of a class and doing a text search on that page
- Once on page, do a text search for the desired method (if they do not use text search, see weak indicator);
- Only looking at the preview of each search result on the page as their source of information without clicking into any search results

- Slowly browsing the results, usually spend longer time on the search engine and scroll further down for more information then simply deciding the results are not useful and thus leave the browser or do another search
- Often we can see what they are reading by noticing when participants move the cursor to trace through the text while reading

Weak:

- Using broad search term and navigating to online documentation of a class and go directly to the desired method
- Though it may appear to be the participant is simply scrolling through a list of available methods, if they scroll faster (instead of slowly browsing) and stop at where they found the method, it is likely that they are using resources for syntax.

Indicators of using resources for syntax when they stay within jGRASP

Students open up another existing file (which generally is previous homework or examples). (Not able to identify indicative behaviors for how they use the file yet, see [Behaviors before going to resources] [Time spent on the resource] and [Behaviors after coming back to the solution file] in order to make a decision)

Behaviors after coming back to the solution file

Strong:

- Pasting up to two lines of code
- Few character or single term edit at the line they were working on before leaving the file for resource

- Such behavior happens when students double check that they wrote the right syntax, which means the resource they looked up is related to what they are typing right before they left the file they were working on
- Adding a package (line starting with "import") to the top of the editor

Middle:

- Editing current line: finishing the line of code that the participant left off before going to resources or editing within the same line, and the resource they look up is relevant to the line they are editing.

Weak indicator:

- Small edits on previous lines (a few character or single term)

A.1.3 Using resource for approach

Purpose of using resources for approach

- Participants use resources for approaches to help them determine how to solve the problem. The approach could be for the whole problem or for a subproblem. Here is a list of examples of using resources for approach:
- Look up existing solutions to help approach the whole problem
- Look up a list of available methods from a class to look for a potential method to use
- Look up how to implement a certain type of data structure or algorithm
- Look up how to implement recursion
- Recall that the problem is similar to an example or a practice problem from class, look up in course materials or existing java files
- Looking up methods online and examples (either online or in files) to understand a conceptual mistake after getting an (simple) error.

Behavior before going to resources

Strong:

- Occurring after stopping to think
- Occurring before starting implementation for the whole problem.

This label often occurs when participants are figuring out approach / planning, which often happen at the beginning of the session, but can also occur when they change approaches during implementation

Middle:

- Occurring after a pause in typing that is generally 15-30 seconds, even if it is not counted as stopping to think

Behaviors while using resources

Time spent on resources for approach If the situation fully fits the description for special cases, we would use special cases for time on resources instead of total time for evaluating time indication. If not, use total time.

Total time

Strong:

- Spending more than 1 minute on resources

Neutral:

- Spending 30 seconds - 1 minute on resources

Negative:

- Spending less than 30 seconds (less than 20 seconds is extremely negative)

Special cases for time on resources

Strong:

- Clicking into two search results (or more) from the same search engine and search term without leaving the browser and spending at least 30 seconds on both (or at least two) of them

Middle:

- Clicking into more than two search results
- Regardless of time spent on each result, but will be a strong indicator if total time exceeds 1 minute
- Different search result could be under the same search engine and search term OR indistinctly search terms OR search terms that are highly similar (only off by one or two words, the overall meaning of the search term is still the same or very similar) using more than one type of resource without going back to the file they are working on, and spending at least 30 seconds on one type of the resources

For leaving jGRASP IDE entirely

When using chrome, it is indicative of using resources for approach when students do the following things: *Strong indicator:*

- Going to sites that are articles and blog post about solutions
- Terms that we are likely to see in Google searches: words used in the assignment, for example: ”duplicate” and ”remove” (although remove might not be good enough, so maybe remove duplicates or removing duplicates is better) ”permutation”
- Asking a broader question about the problem solution or a particular aspect of the problem (examples: “how to get the next word java” ”how to remove duplicates strings from array java” ”reading through a document word by word in java”)
- Terms about problem-solving techniques and algorithms, for example: “recursion”

Middle indicator:

- Using provided examples on website such as StackOverflow or on their course website with example problems as reference for an example structures

Weak indicator:

- Using broad Google search and navigating to documentation of a class and appearing to be slowly scrolling through instead of looking for a specific method AND/OR text search "method" within the documentation

NOTE: Broad google search terms are NOT indicative one way or the other unless we analyze how they navigate the site (like scrolling). Browsing online course material, especially explanations of concepts

For within jGRASP

Students will open up previous homework or examples.

Strong:

- Freezing the screen on one part of the existing file for a longer period of time (for reference: more than 20 seconds) without scrolling up and down

Middle:

- Opening up previous homework or examples, slowly scrolling through the editor to see how they implemented a similar approach in the past

Behaviors after using resources for determining approach

After coming back to IDE in the file they are working on, we could see one of the following behaviors to help identifying using resources for approach:

Strong:

- Pasting in a large chunk of code from a resource and editing it afterwards to fit the problem they are working on
- Label - stopping to think
- Pausing shortly without typing anything in the IDE and doing another search instead

Middle:

- Pasting more than two lines of code
- Starting implementation for the whole problem
- Label - changing approach (see description for changing approach)
- Toggling back and forth the same resources while implementing

Weak:

- Starting typing on a new, empty line, and continuing with the current approach

A.1.4 Using resources in general

This general using resources label is used when the behaviors of using resources is not indicative either syntax or approach, or if the purpose of using a resource is not clear and the indicators are not clearly for either one.

Situations where we can not determine if participant is using resources for syntax or approach, thus label as general

- Student looks up resources before starting to implement, scrolls a while, finds something and then starts to implement. (were they scrolling and looking for it specifically or were they scrolling until they found the right thing and thus needed an approach and found it so started implementing

- 2.10 scrolling in chrome behind IDE
- When a google search does not seem relevant to what they are doing, such that we don't know if they were looking up something for syntax or approach. For example: googled "char to int", clicked into one for a few sec, hit back, clicked in again for a few sec, then googled integer value for char java, clicked into one for a few sec, hit back, clicked into another one with an ascii table
- If we really don't know how to tell their original purpose for going to a resource.

Purpose of using resources that is not for syntax or approach, thus label as general

- Use resources for jGRASP features.
- Sometimes participants seek help on how to use certain features of jGRASP on chrome, such as debugger, instead of help on solving the programming problems. Therefore, the following behaviors should not be seen in the moments of using resources:
- Google search terms that contain "jgrasp"
- Looking at online jGRASP help
- Look up meaning of a word from the problem description or resources
- This also includes if they are searching for the translation of a word.

Behaviors that are considered "neutral" as they are not indicative of either syntax or approach (although they are not necessarily indicative of general)

- Using Google search for example syntax from websites like Geeksforgeeks, W3, stackoverflow, and the search term is unclear;
- Opening course slides without using text search for a specific method

- Opening course website and flipping through a couple different pages on the website without spending noticeably longer on any of them

A.2 Getting simple errors

Explanation of the label: These are characterized by errors that a student could easily overlook and think they should be easy to fix, although it may take a long time for a student to identify the cause of the error. Simple errors are distinguished by forgetting something and not a conceptual mistake/misunderstanding. Simple errors must be compiler errors. No runtime errors count as "simple" errors. An "error" in their program logic that causes unexpected runtime behavior also does not count as an error here.

It is possible that they get multiple errors at a time, when determining if it is a "simple error", we should use the first error in the console and on the highlighted line suggested by the IDE, because that is usually what they tackle first.

Errors from testing in the interactions or from testing in a separate file will also be evaluated for simple java error and java error labels.

Common examples of simple errors

- While watching the participant programming, we could tell a compiler error is a simple error if the cause of the error is one of the followings:
- Forgetting to import a directory
- Declaring more than one variable/ function with the same name
- Missing parenthesis/ semicolon/ square bracket
- Mismatching curly brace(s)
- Forgetting to declare a variable before calling it in the program
- Violating variable naming rules

- Incorrect syntax for method calls, such as mis-spelling method names or forgetting to add ()
- Incompatible type conversion, such as data type of a variable and return type of a function
- Using incorrect variable names due to mistakes like mis-spelling
- Incorrect syntax for initializing a variable

Error messages from an simple error

Here is an example list of exact error messages that often are the results of simple errors:

- ”error: ‘ (‘ expected” ”error: ’)’ expected”
- missing parentheses
- ”error: ‘ ;’ expected”
- Missing semicolons
- violating variable naming rules
- “error: not a statement”
- violating variable naming rules
- ”error: bad initializer for for-loop”
- Incorrect syntax for a for loop
- “error: xxx is already defined in yyy”
- Declaring more than one variable/ function with the same name
- “error: incompatible types: ...”
- Return statement does not match with function header

- Passing in wrong data type for a function input
- Declaring a variable with incorrect data type
- “error: reached end of the file while parsing”
- Missing closing curly brace(s) at the end of the program
- “error: cannot find symbol”
- Forgetting to import a directory
- Forgetting to declare a variable before calling it in the program
- Using incorrect variable names due to mistakes like mis-spelling
- Incorrect syntax for method calls

A.3 Getting Java error

Student gets runtime error, including runtime errors in both the interactions and in the editor, or student gets compiler error, and the cause of the compiler error does not fit into the definition of simple error.

Errors from testing in the interactions or from testing in a separate file will also be evaluated for simple java error and java error labels.

Examples include:

- compiler error:
  - ”error: class, interface, or enum expected”
  - “error: java: file not found”
- runtime error:
  - “error: array index out of bound”

  - “exception in thread ”main” java.lang.StackverflowError”

However, even if the cause of the compiler error fits into a simple error, if the participant never fixes the error and moves on and avoid the error, it should be labeled a Java Error and NOT a simple error. This is because the participant doesn’t figure out what the root of the error is, and thus does not know that it is simple. NOTE: if the participant runs out of time and that’s why they can’t fix it then it can not be determined.

Common examples of compiler errors that participants label as “simple” but sometimes fail to fix:

- “Error: illegal start of type” or “Error: illegal start of expression”
- Such errors usually come from mismatching curly braces in the program, and we have found that some participants had a hard time identifying that missing curly braces is the cause. Though they might label it as a simple java error, if they did not succeed in fixing it, we would use a java error label in this situation.

A.4 Struggling with error

The error the participant is struggling with must be either compiler error, runtime error, or unexpected program runtime behavior. In order to qualify as struggling with error, these behaviors must happen directly after getting a java error, simple java error or unexpected runtime behavior.

Behaviors after getting errors and struggling with errors

NOTE: It does not count as struggling with errors if they fix the error before stopping to think, even if it is a simple error and they do not run the code to get rid of the error (for example, if they put a semi colon in where it was missing but do not push run because they know it is fixed).

Strong indication:

If we see one of the following behaviors, we could be certain that the participant is struggling with errors.

- (for java or simple java errors) Idle time with jGRASP or the problem description in focus (no activity) for 1.5 minutes (will get stopping to think label).

- Idle time can include scrolling and looking at the errors, even though that is not included in stopping to think

- Allowed to make changes to whitespace within time as long as they are only a minor edit (not an attempt to fix the code). Edits can only be erasing or adding whitespace.

- Using debugger or interactions with an active error in the console or after unexpected runtime behavior

- (for java or simple java errors) Running or compiling code and receiving an error for the same issue at least three times, with some changes being made in between. Must qualify as in the same error cycle. If it is less than three times, see the middle indicator.

- (for unexpected program runtime behavior) Commenting out test cases or commenting out the lines that call the main function in public static void main(String[] args)

Middle indication:

These behaviors do not qualify for the moment on their own. If we see at least two of the following behaviors or one of them in combination with the behaviors mentioned in the strong indication section, we could label the moment for struggling with error.

- Using resources for syntax or approach after getting error(s)

- Temporarily commenting/ deleting out a piece of code and rerunning the program **in order to understand which part of the code is causing the errors

- “Temporarily” means we should see commenting, rerunning (or recompiling), and uncommenting happen consecutively during the time they are working on the error(same for deleting, rerunning, and restoring the piece of code that has been deleted). The time span of

temporarily commenting/ uncommenting is usually under 2 minutes, but it depends on a participant's overall implementing speed.

- If the participant does not uncomment/ restore, it is likely that they are "changing approach". See the description for changing approaches

- Rerunning (or recompiling) code without making any changes.

- Making changes to the code, and then rerunning (or recompiling) it but didn't fix the error. Changes could be any size, but changes should not be directly commenting out or deleting the lines that have errors. In order for it to count as rerunning (or recompiling) and in the same error cycle, make sure that it qualifies as an error cycle (see definition).

Weak indication (General behaviors):

It is common to see at least one of the following behaviors while participants are struggling with errors, but these behaviors should not be used as an indication of the label struggling with an error.

- Slowly scrolling through the console to read all the error messages

- Slowly scrolling through the IDE to examine the code, including their own implementation and the provided starter code

- Slower pace of typing speed

- Stopping to think after getting error but under 1.5 minutes (over 1.5 minutes is a strong indicator)

Moving on without fixing the error

Sometimes participants might continue implementation for the problem without fixing the current error, or without running the code to confirm the error is fixed. If the participant has not rerun the code after making changes and is typing/ editing the code regularly on lines not suggested by the

error messages (similar to the pace before getting errors), the participant is no longer struggling with the error and has moved on to continue implementing.

A.5 Writing a plan

Could occur before or after implementation has begun. Participants externally brainstorms or plans. This is in the form of: writing in the IDE or writing on paper/ipad or notepad on computer.

Within the IDE

This is what the comments should be that count as planning:

- Writing comments that describe the steps that they think they will use to approach the problem- either inside or outside of the main function (which is the provided function with /*your solution goes here*/)

- Writing comments in the middle of a function/ loop body as a placeholder for coming back to finish code implementation later. The function/ loop body does not have to be empty. (our examples are between 1 and 4 lines). The content of a placeholder comment should be a direct instruction of what functionality the participant is planning to implement at that spot

- Writing comments before a loop (could be a nested loop) or the header of a function defined by the participant that details the participants' plan for what they will put in the loop/function or their brainstorming to figure out what should go in there.

Note: When they are writing a plan in comments, we will only re-evaluate the moment when they type code in between writing comments. Other situations such as stopping to think, using resources, etc., will not affect the current session of writing a plan in comments.

Specific indicators to help identify comments that are planning:

(Note, purpose of comment should always be priority, but these indicators can help to determine the purpose of the comment) When watching a participant program, we found that the comments

that count as writing plans often are:

- Are at least 2 lines long
- If the comment is only one line:
  - Should be at least around 5 words
  - Both the line right above and below should be empty at the time when the participant writes the comment OR
  - The line of comment is right above an empty helper function (empty = they only wrote the declaration) or an empty loop (empty = only set up the conditions)
- include content that is:
  - direct instructions on what to do in code
  - a combination of written language and pseudocode/ methods.
- are ALWAYS independent and NOT inline

Comments that should NOT count as planning:

- Commenting out existing code
- Single line comments directly above functions/loops that have code in them (it still counts if the content is only comments) This is because the single line comment is usually documenting the purpose of the functions/loops after they start to implement the functions/loops body.
- Sample output of the program mentioned in the problem description (more likely to see before starting implementation)
- Instruction for the problem based on the problem description

  - Usually appears at the first line in the IDE and outside of the function they are going to work on
  - The participant could directly copy and paste from the problem description or write in their own words
  - Usually happens in the beginning phase of programming when the participant only has little or no implementation

- Reminder: comments that are used to remind themselves instead of part of the planning, such as “needs RETURN”

- Documentation of existing code, usually are inline comments or on the line directly above the code

Outside of IDE

Participant could be writing plans on notepad or other text editor on computer OR writing plans on paper/ ipad. However, since we are not able to see if they are writing plans on paper/ ipad based on their screen activity, either in detection or by human watching, we will not consider such behavior for evaluating writing a plan label.

If we were considering content in an outside text editor, this is how we would identify if it was planning:

Include the content that is:

- direct instructions on what to do in code

- a combination of written language and pseudocode/ methods.

Exclude the content that is:

- Sample output of the program mentioned in the problem description (more likely to see before starting implementation)

- Instruction for the problem based on the problem description. The participant might directly copy and paste from the problem description or write in their own words
- Reminders for what to do that are not planning

A.6 Stopping to think after starting implementation

Time indicator for stopping to think

Student is idle on the screen for at least 25-30 seconds (although generally over one minute) AND is a lengthy pause relative to the pace by which they are programming (if they are moving quickly, a shorter pause will count compared to if they are moving slowly).

Situations that require longer pause

At the beginning phase of implementation

If the participant is still at the beginning phase of implementation, the idle time must be at least 45 seconds in order to qualify for stopping to think. This is because students may go back to the problem description to try to understand the problem and that is not stopping to think about implementation. Beginning phase of implementation means the participant only has little implementation, such as initializing one object, and not yet starts building logics or functionalities. In general, implements less than 3 lines of meaningful code.

Before going to resources for approach

If the participant pauses before going to resources for approach, the idle time must be at least 30 seconds in order to be qualified for stopping to think. This is because they may just be thinking about how they want to use the resource and that is separate from stopping to think independently.

Coming back from resource

If the participant is scrolling around on the screen initially, this may be to get their bearings again (especially after using a resource for a long time) and the time spent scrolling does not count towards stopping to think.

Getting simple java error or java error

Time spent actively scrolling through the console output or the run I/O doesn't count as stopping to think if they are scrolling to read the content and not aimlessly scrolling around the output. Additionally, the idle time must be at least 30 seconds after getting an error. Also 30 seconds if the debugger is open.

Unexpected runtime behavior

Time spent actively scrolling through run I/O doesn't count as stopping to think if they are scrolling to read the content and not aimlessly scrolling around the output. Time spent toggling between run I/O and problem description doesn't count as stopping to think.

During writing a plan

If the participant pauses while writing a plan, either in the IDE or in another text editor on the computer, the idle time must be at least 30 seconds in order to be qualified for stopping to think. Stopping to think CAN occur while participants are in the process of writing a plan.

Activity on the screen that disqualifies stopping to think

During the pause (i.e. not typing code or no activity in the IDE), it does not count as "idle time" if the participant is:

- Looking at any resources
- Writing plans
- (Slowly and) steadily scrolling through the IDE to read the starter code, or looking at the provided test cases
- Scrolling in the console to read the error messages. It is very likely to have mini pauses in between scrolling. If they pause for longer than about 5 seconds, the time will be counted into idle time.
- Making changes to the code

  - Inserting or deleting empty lines does not count as changes to the code
  - Exception: if they already have some initial implementation (at least 2 meaningful lines of code) by the time they pause, they can make small edits, generally up to 10 characters or add/ finish up to one line of code at the line they stop, as long as there are at least 20 seconds of idle time before and after the changes and still qualify as stopping to think. Each set of changes should not take longer than 10 seconds.

NOTE: Writing comments that do not count as planning does not disqualify stopping to think.

Participants might toggle between jGRASP and the problem description a couple of times while stopping to think. Therefore, during a pause, they could have jGRASP or the problem description in focus. In focus means the "selected window". If there are multiple windows on the screen, it should be the window on top. If it is split screen, it should be the window that the participant last interacted with.

Notable behaviors before stopping to think

- Label: getting simple java error
- Label: getting java error
- Label: unexpected runtime behavior

Notable behaviors after stopping to think:

- Label: Using resources for approach
- Label: writing a plan during implementation
- Label: changing approaches (realizing the current one will not work and thinking about a new one)

A.7 Changing approach

Changing approach could be trying a new approach, abandoning an idea or going back to the old approach. We will only include situations where changing approaches happens in their implementation in the IDE and not a change in their plan (see Note A)

Preconditions

Preconditions are the labels that happen before the participant starts changing approach. To determine if something should count as a precondition, it should be while they are working on the same topic. So if they get an error and solve it and move onto something else, this error is no longer a precondition, however, if they get an error and then use resources for five minutes and then change something, the error can count as a precondition. Generally, this happens within four minutes of changing approaches, but that may be longer if the participant spends a long time on something like using resources.

At least one of these preconditions must be seen as part of the qualification for the moment:

- Label - Stopping to think (very likely to see)
- Label - Using resources, could be for determining approach or syntax (common to see)
- Label - Writing a plan during implementation (less likely to see)
- Label - Getting Java error: must be runtime error that comes from the implementation, not error from testing in the interaction or a separate file
- Label - Unexpected runtime behavior

Behaviors after preconditions

The following behaviors could indicate changing approach and are categorized into three sections (strong/ middle/ weak) based on the level of indication. If more than one of these behaviors happen, they usually occur within 3 minutes of the other. In some cases, it could go above 3 minutes if

stopping to think or using resources happens in between. They can indicate the same "changing approaches" label if the participant is still working on the same topic.

Strong: the behaviors that fall under this section are enough to be qualified for the label alone without being in combination with other behaviors after the presence of at least one of the preconditions.

- Inserting at least 5 new, empty lines to push down the old existing code, AND starting new code that looks similar to the one that is being pushed down (for example method names or structure). (This happens when participants program a new idea before getting rid of the old one)
  - New code must not be comments
  - Old code (the part that has been pushed down):
    - Must be at least 3 lines of "meaningful code" OR has the structure of loop(s) and/or conditional statement(s) created but not filled in.
    - The "old code" ends at the end of the function the participant is currently working on, which could be the main function they are asked to work on or a function they defined on their own
- Changing the structure of the code from using a for loop to using recursion, or the other way around
  - Recursion is implemented by a function calling itself, so it is generally a helper function
  - Likely, this will look like a participant deleting a for loop and moving part or all of the contents of the loop to the recursive helper function and then calling that function from where the loop was.
  - This behavior is more likely to take longer to implement than other changing approaches indicators.
- Erasing all the implementation and then starting over

  - The implementation must have some structures (for reference: 3 meaningful lines of code) or is a chunk they have been working on for a while
  - Must have counted as starting implementation.
  - Does not require any precondition

Middle indicators:

The behaviors that fall under this section are commonly seen when changing approaches. They independently qualify for the label if at least two of the preconditions happened. Otherwise, one of these behaviors must be in combination with at least one other behavior from strong/ middle/ weak sections to indicate the label.

- Commenting out or deleting at least 3 meaningful lines or 50
  - Does not count if the code is commented out and then reinstated within one minute IF there is a code run or compile between those two things. See Note B for further explanation
  - Might only delete one or two lines of code at a time in the process of changing approach, in this case, the lines of deleted code can be accumulated over time as long as it is within the 3-minute period

Weak indicators The behaviors fall under this section are either not as common as the two behaviors in the middle section or could easily be mistaken as simply editing approach. Therefore, the behaviors in this section should only be supplements and must be in combination with other behaviors in strong/ middle sections. We may label it as changing approaches if there are two weak behaviors AND two preconditions.

- Changing conditions of loop(s) and/ or if statement(s)
- Changing the output data type of a helper function defined by the participant
- Erasing/replaces all of the content within a loop or conditional statement.

- Uncommenting or restoring code
- Implementing a new type of data structure at the top part of the code
  - Must already have implemented some code before this, specifically at least 3 meaningful lines or has a structure of loop(s) and/or if/else statement(s)
  - Example: hashset, arraylist

Negative Indication

- Run/ compile the code before finishing implementing for the whole problem. It is unlikely to see the participants compile the code either right before or after changing approaches if they have not yet finished implementing the whole problem. It is more likely that the participant is testing to see if they fixed an error instead of changing approaches.

A.8 Unexpected runtime behavior

Unexpected program runtime behavior happens when the program successfully runs with no error but the output is different from what the participant expects. This usually happens when they think they are done with implementation for the whole problem, but the output they get is incorrect. Although less likely, it is still possible to see unexpected runtime behavior for partial test of the program instead of the full program.

Indication of the moment

Strong indicator

- Fixed multiple compiler errors before the program successfully runs with no error, but fails the tests. Errors could be simple java errors or java errors, but must not be from testing in the interactions or in a separate file. It is common to see the participant try to compile the program a few times throughout the programming session to check for/fix syntax errors, yet they do not hit run until they think they are done with implementation. It will still be

a strong indicator of unexpected runtime behavior if they actively fixed errors that come up throughout the programming session and then ran the code and it did not pass the tests.

- Changed approaches after getting the previous unexpected runtime behavior label, run the program, and the program output is incorrect
- User has to end the runtime. Will say: "Process ended by user" in run I/O
- (For full program test) The first time the participant successfully runs the program for a full program test but receives incorrect output
- (For full program test) Ignore the parameter passed to the function and use System.in in the implementation instead, test the program with their own input in Run I/O and get incorrect output

Middle indicator:

Need to make sure for middle indicators that they were not expecting the output that they got and were trying for something else.

- Made changes to the program after getting the previous unexpected runtime behavior label, run the program, but the program output is incorrect.
- Changes might include adding new variable(s), adding new conditional statement(s), deleting lines, etc., but not qualified for changing approaches label
- If they were not expecting the full program to run correctly, and they were expecting this part of the program (running a partial test), then it is not unexpected runtime behavior. Additionally, if they fix something in the code that they know will fix a mistake but not solve the full problem, it does not count as unexpected runtime behavior.

Negative indicator

- Run the program at the beginning, even if they get no error and an output. At the beginning of a programming session means when the participant has not yet or just started to implement.

Some participants would run the program at the beginning to check if the starter file is able to successfully run and compile on their computers, and thus they are expecting to see empty/ incorrect output or failed test cases messages.

- Run the program when the main function is empty, even if they have worked on other self-defined function(s). In this case, if they hit run, they are just checking if there is any syntax error in the self-defined function(s).
- It is unlikely to be this label a second time if they rerun and get that same incorrect output as the previous run, especially if there are minimal changes to the code between runs (only different by a few characters), as the participant is expecting the incorrect output that time..
- The participant makes no changes to the program since the previous unexpected runtime behavior label(s) (or added and removed changes so that the program is exactly the same as the last unexpected runtime behavior). This is because the participant knows the program will generate the same incorrect output, and thus it is not an unexpected runtime behavior. For example, after getting unexpected runtime behavior, the participant made changes to the program, but the changes generate runtime error. The participant removed the changes, reran the program, and got the same incorrect output.
- If the only change between runs is that the participant adds or deletes console outputs (system.out. . .) statement, it is generally not an unexpected runtime behavior moment.

Behaviors after the moment

- Strong indicator
- Label - changing approach
- Label - stopping to think

A.9 Glossary of terms and instructions

A.9.1 Term: Main Function

The function that the participant is asked to work on, which is the function that contains "/*your solution goes here*/". Not referring to public static void main(String[] args).

A.9.2 Term: Implementation

Defining the term: starting to implement.

Starting to implement occurs when the participant adds any characters to the IDE that are not:

- comments
- solely import statements at the top of the IDE (ex: import java.utils.*)
- exact same code they delete from the starter code

Characters must persist in the IDE for generally at least 30 seconds or enough to such that it isn't immediately deleted.

Additionally, if at any point in implementing, a participant erases all of the executable code that they added (not code present in the starter file), they go back to the "before implementing" stage. When they add characters again (abiding by the same criteria as above), it counts as starting to implement again

A.9.3 Term: Error Cycle

When participants rerun their code after getting an error, in order for it to count as in the same error cycle, the following qualifiers must apply:

- The new error(s) must be consistent with the original error as either compiler error or runtime error

- Any of the original errors should not be solved. The participant may have introduced new errors so the original error does not show, but the original error should still be an issue (and would come up if there were to remove the newly introduced error. The new errors may be exactly the same, more or different errors from the original error, but the cause of the original error should still be an issue.

Side note: there is an internal hierarchy in jGRASP that determines the order of what errors to display first. For example, the IDE will not display "error: cannot find symbol" before the participant fixes "error: ' ; ' expected" even if both errors exist in the program at the same time. Therefore, simply using error counts is not a reliable way to identify struggling with error after rerunning.

Things to note that start a new error cycle:

- Error count going down (not from introducing a new syntactic error -¿ which generally is only 1 error)
- Errors resolved entirely
- Completely different errors for a different issue (and solved the initial issue)
- Participant moves on to work on something else -¿ could be seen by switching files, commenting/deleting code

NOTE: A compiler error may not stop an Error Cycle for Runtime errors if its syntactical

A.9.4 Term: Meaningful Code

"Meaningful code" is code written by the participant and is part of the participant's actual implementation for the problem. When referring to "meaningful lines" in the context below, we mean lines that are not:

- Blank
- Provided starter code

- Code written for testing purposes, for example, system.out.println()
- Only curly braces
- Comments

If the participant deletes the line in the middle of writing it or after just finishing it, the line will not be counted as a meaningful line as well.

When the participant defines their own function(s) outside of the main function as helper function(s), if the participant comments/ deletes out all the lines in the main function that call the helper function(s), it is the same as commenting/ deleting out the whole helper function(s). Thus, the line count of commenting/ deletion will be the total meaningful lines of the helper function(s).