An Exploration of Student Reasoning about Undergraduate Computer Science Concepts: An Active Learning Technique to Address Misconceptions

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Human-Centered Computing

---

by
Cazembe Kennedy
May 2020

Presented to:
Dr. Eileen Kraemer, Committee Chair
Dr. Lisa Benson
Dr. Kelly Caine
Dr. Murali Sitaraman

Abstract

Computer science (CS) is a popular but often challenging major for undergraduates. As the importance of computing in the US and world economies continues to grow, the demand for successful CS majors grows accordingly. However, retention rates are low, particularly for under-represented groups such as women and racial minorities. Computing education researchers have begun to investigate causes and explore interventions to improve the success of CS students, from K-12 through higher education. In the undergraduate CS context, for example; student difficulties with pointers, functions, loops, and control flow have been observed. We and others have utilized student responses to multiple choice questions aimed at determining misconceptions, engaged in retroactive examination of code samples and design artifacts, and conducted interviews in an attempt to understand the nature of these problems. Interventions to address these problems often apply evidenced-based active learning techniques in CS classrooms as a way to engage students and improve learning.

In this work, I employ a *human-centered* approach, one in which the focus of data collection is on the student thought processes as evidenced in their speech and writing. I seek to determine what students are thinking not only through what can be surmised in retrospect from the artifacts they create, but also to gain insight into their thoughts as they engage in the design, implementation, and analysis of those artifacts and as they reflect on those processes and artifacts shortly after. For my dissertation work, I have conducted four studies:

1. a conceptual assessment survey asking students to "Please explain your reasoning" after each answer to code tracing/execution questions followed by task-based interviews with a smaller, different group of students

2. a "coding in the wild" think aloud study that recorded the screen and audio of students as

they implemented a simple program and explained their thought process

3. interview analyses of student design diagrams/documentation in a software engineering course, tasking students to explain their designs and comparing what they believed they had designed with what is actually shown from their submitted documentation

These first three studies were formative, leading to some key insights including the benefits students can gain from feedback, students' tendencies to avoid complexity when programming or encountering concepts they do not fully grasp, the nature of student struggles with the planning stages of problem solving, and insight into the fragile understanding of some key CS concepts that students form. I leverage the benefits of feedback with guided prompts using the misconceptions uncovered in my formative studies to conduct a final, evaluative study. This study seeks to evaluate the benefits that can be gained from a guided feedback intervention for learning introductory programming concepts and compare those benefits and the effort and resource costs associated with each variation, comparing the costs and benefits associated with two forms of feedback. The first is an active learning technique I developed and deem misconception-based feedback (MBF), which has peers working in pairs use prompts based on misconceptions to guide their discussion of a recently completed coding assignment. The second is a human autograder (HAG) group acting as a control. HAG simulates typical autograders, supplying test cases and correct solutions, but utilizes a human stand-in for a computer. In both conditions, one student uses provided prompts to guide the discussion. The other student responds/interacts with their code based on the prompts. I captured screen and audio recordings of these discussions. Participants completed conceptual pre-tests and post-tests that asked them to explain their reasoning. I hypothesized that the MBF intervention will offer a valuable way to increase learning, address misconceptions, and get students more engaged that will be feasible in CS courses of any size and have benefits over the HAG intervention. Results show that for questions involving parameter passing with regards to pass by reference versus pass by value semantics, particularly with pointers, there were significant improvements in learning outcomes for the MBF group but not the HAG group.

Dedication

This dedication is written in two parts. Family: Present and Future.

Present

To my current family, I love you all and literally could never have been in the position to obtain this degree without you. Starting with my parents, who provided me the genes (nature) and environment (nurture) to become the person I am today, I thank you both for being there for me. I've always (or since I could really form beliefs) believed that everything happens for a reason. I have been extremely fortunate in my life and a lot of that stems (no pun intended) from my parents. To Ayanna, my 18-month older twin. I love you, sis. I have been and will always be extremely proud of everything that you accomplish. I appreciate so much that we have a great dynamic and even though we're close in age and in similar fields, have never felt like we were competing against each other. Our achievements and successes can be shared with each other knowing that we'll always love, support, and protect our sibling. My grandparents, who are not with us anymore, but gave wise advice and provided many happy moments for me, not to mention shaping the people who shaped me (my parents). Also the cousins, aunts, uncles, and every other family member who is pivotal in my life. Finally, in the present section I would like to dedicate this to my chosen family. Anyone who knows me knows that I consider my best friends my chosen family. You guys have been with me anywhere from 7-23 years, and I know I can always count on you to tell me what I need to hear, whether or not that's what I want to hear. I love you guys and cherish the bonds we've built.

Future

This part is a little bit more difficult, because I don't know exactly who I'm writing to. What I do know is that for the last half a decade or so, most of the positive things I've done in my life to achieve success, gain money, etc., have been because I know that sometime (hopefully soon), I will

have a family of my own and they will become my world. To my future wife, I hope you're enjoying life to the fullest, as much as you can during these quarantimes (intentional misspelling). I want you to know that all of my life experiences and opportunities have been leading me to and preparing me for you. I enjoy being semi-dramatic, so I will probably let you read this sometime before the wedding, or maybe even put it in my vows, but just know that through any fight, disagreement, or simply unpleasant times, I chose you for a reason and I will continue to choose you. There was a reason I waited so long to get married and was unwilling to settle. I needed to find you, and am so glad I have. Now, the important part (love you dear :))....Our future children. Hey! I bet you would not have imagined your dad thinking about you so far ahead of your birth/s. I won't say your names because it would be awkward if they don't end up being true. These almost 300 pages of reading are what I'm dedicating to you (super cool, right?). I have no expectations of you reading them all, but when thinking about important people in my life, I just could not include you in this section. The thought of us growing together as a family, having lame traditions that you might pretend to dislike but know you'll love...It's something that has motivated and continues to motivate me everyday. I imagine I might tear up reading this years from now, as I'm currently on the verge of it just thinking about the great times there are to come. Just be your best, which I know already has the potential to be better than me. Your mom and I won't be perfect parents, as no one is, but love, support, honesty, and trust are things I can guarantee we will provide you through this process. If you ever need some encouraging words, maybe these will help. I know you've probably heard the phrase "You can do anything!!" Well that's untrue and not realistic. Some things are just physically or practically impossible to do. BUT, what you can do in the time you have on Earth is incredible and please make sure that the things you do make you and the people you love happy. If you live life like that, it will be as full as I've seen it.

With love,

-Dr. Dad

P.S. All parents have a favorite. We love you equally, but depending on the time, we might like one of you a little more :)

Acknowledgments

I would like to first acknowledge my committee for all of their suggestions, feedback, and support throughout this process. Dr. Kelly Caine was instrumental and very helpful in suggesting an evaluative study to round out my dissertation and also provided useful feedback for the quantitative analysis of my evaluative study. Dr. Murali Sitaraman has been familiar with my research for years now and helped simplify the initial study design from when I proposed my evaluative study. He also has given me multiple opportunities to present research at multi-institutional research groups he coordinates and within his shared lab with Dr. Eileen Kraemer. Dr. Lisa Benson has provided invaluable expertise in discipline-based education research, as coming into my program, I had the science knowledge, but had never conducted education research. She has also been a professional and personal mentor, "adopting" me into Clemson's Engineering and Science Education department and giving me advice for future goals while helping me attain those goals by connecting me to people to help further those goals. Lastly, the chair of my committee, Dr. Eileen Kraemer, who became the person I interacted with the most towards the end of my PhD journey. We have different, but complementary styles and I cannot thank her enough for her assistance and support throughout all six of these years. I was very fortunate to have met her when we arrived at the university the same year.

I would also like to acknowledge the people outside of my committee who have helped with my research and growth through this process. Dr. Bridget Trogden was one of the invaluable connections I made through Dr. Lisa Benson. I could potentially write another conference paper on the awesomeness that is Bridget. The opportunity to work as a graduate research assistant with her for 1.5 years has opened up so many doors and I know we have made an impact on education policy. It's through your tenacity and support that I found the connection for my first career position post-graduate school. In short, you're great and stay that way. Matthew Re helped with transcribing the

data for my evaluative study. Dennis Lee helped introduce me to qualitative coding software while I helped code data for his dissertation and just is awesome. Glad we're "graduating" together! Aubrey Lawson was one of the qualitative coders on my team for my research and has co-authored conference papers with me, been supportive during international conference travel, and just all around been a great friend in our research group. Finally, I would like to acknowledge that research group. Juan, Aubrey, Megan, Dan, Sami, Nikki, and Olivia have all been a great help whether it was advice on research or just having people to share the graduate school experience with or relax and de-stress when necessary. Thank you all.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

To address the impact of rising enrollment of the computer science (CS) major on the ability to provide high quality education with limited resources, schools have adopted many approaches. One of these approaches is using autograders to provide feedback in a more efficient manner that saves instructors time and resources. However, the use of such automated feedback may fail to provide the pedagogical benefits of the quality feedback that is possible with lower enrollments[10, 72]. While quality feedback seeks to guide students to knowledge of related concepts and address misconceptions, autograder feedback generally merely compares executed code to test cases provided by an instructor[57], limiting it to providing confirmation feedback[120]. In this dissertation, I describe several studies that I have conducted to uncover student misconceptions and a peer feedback intervention that I developed and evaluated that employs active learning and involves a structured dialogue between students. I first conducted two studies to identify misconceptions students have about key computer science concepts and gain insight into the cause of those misconceptions. Those studies showed a key element of the value of feedback, so I developed an active learning technique based on misconceptions and using peer feedback through structured dialogue to evaluate the results of the formative studies. The prompts that guide the dialogue are developed based on these misconceptions and related work on learning, particularly how novices learn. I then compare the outcomes associated with that technique with a protocol simulating automatic graders that would be classified as active learning, but not allowing students the benefits of constructing their own knowledge or working together to interactively construct knowledge[31]. This human autograder approach was used to control for the delivery element of the feedback, as the misconception-based feedback was

provided by a human peer. In my work, I present an active learning technique that I developed, named misconception-based peer feedback (MBF), to offer a low resource method to improve student learning outcomes and that can be used even as computer science course size continues to increase.

Computer science (CS) is a challenging major for undergraduates[58, 142]. Retention rates are low, despite the ever-growing demand for employees in the field. In 2012, a presidential report discussed this problem, and particularly noted the lack of underrepresented groups (URGs) such as women and racial minorities[142]. Integrated Postsecondary Education Data System (IPEDS) data reports representation of under 10% for Black and Lantinx students earning bachelor's degrees at both doctoral and non-doctoral-granting institutions[3]. The same report shows under 20% for women earning computing bachelor's degrees. Since 2013, the National Science Foundation has developed a "CS 4 All" initiative to attempt to address this issue[70], offering millions of dollars for projects geared towards involving all K-12 students in CS education. Retention rates are also found to be lower with URGs, with Black and Lantinx students having lower retention rates as well as women compared to men transfer students[3]. An important part of the process of increasing retention rates and students successfully learning CS is understanding the obstacles students face while trying to learn the subject.

These efforts to increase diversity in computing coincide with rising CS enrollment numbers. From 2012-2015, CS enrollment more than doubled while Black or African American students only increased 0.5%[138]. This same report discusses methods to address rising enrollments such as enrollment caps or tightening admission requirements, which can have negative impacts on increasing diversity within the field[138]. Tightening admission requirements presents issues such as institutions over-relying on past measures of success (SAT or ACT). Performance on these exams has been found to correlate with socioeconomic factors[76] and URGs have reported lower scores, possibly due to disparities in opportunities available to different populations[138]. Multiple pedagogical suggestions for improving diversity have been proposed, including alternative learning environments[180], collaborative problem solving and interdisciplinary projects[154, 164], and service learning and real world context in the curriculum[16, 46].

Many of the issues involved with addressing enrollment deal with institutional resources and constraints. The National Academies Press (NAP)[138] calls faculty and professional staff the "most significant resource constraint in CS departments." The authors mention how efforts for routine class management increase with enrollment, and some of these efforts do not easily scale. One increasingly

popular approach to address rapid increase in enrollment rates is automated grading of homework, assignments, and tests, providing instant validation of correctness and reducing the amount of time and effort grading and feedback can require of instructors. Research suggests that such instant and automated feedback might be insufficient to address student needs, because introductory students may produce correct output without fully understanding the concepts they are implementing[57, 213]. When students receive positive feedback on implementations that rely on misconceptions, this can lead to a hardening of those misconceptions[105], which can do more harm than good in the formation of appropriate mental models. Further, such automated approaches remove the human element and do not promote the discussion and reflection that are seen in instructor-student or student-student interaction. In my work with misconception-based feedback, I seek to address this issue by providing an intervention that allows students to give quality feedback to each other through structured prompts promoting discussion of and reflection upon common CS misconceptions.

My work is motivated by my personal experience of attending a Historically Black College/University (HBCU) as a CS major for my undergraduate degree. I have seen firsthand the under-representation in CS and the problems that exist in presenting curriculum in a way that is digestible to students while still covering all of the necessary content. This motivation led me to work in the computing education research (CER) field, with particular interest in active learning, as it has been shown to benefit all students, but especially to benefit students from underrepresented groups[142, 82, 126, 73]. Research has shown that stereotype threat[190] and false beliefs around CS identity[11] can have negative impact on academic performance[189, 190]. Despite these false beliefs such as "Computer science is for nerds" or "Computer science is for math whizzes[11]," evidence shows that URGs can achieve and excel as well as the majority[151, 121]. With respect to retention, work has found that lack of motivation and time due to underestimating the difficult course content of CS are major causes of students not staying in the major[107]. A presidential report on STEM education[142] found that students have higher retention rates in STEM if they can successfully navigate the introductory years, and research has shown that many problems in those first years of CS center around misconceptions[148, 201, 132]. I seek to address misconceptions for all students with the hopes of allowing more students to matriculate through the CS degree and help close the achievement gap that URGs still face.

The research questions that guide my overall research are:

RQ1: What insights can human-centered approaches examining code comprehension and code im-

plementation provide into student conceptions and misconceptions about key undergraduate computer science topics?

RQ2: What explanation for these misconceptions can these approaches bring?

RQ3: What can be done to address these misconceptions?

Three formative studies were conducted to address the first two RQs and an evaluative study of an active learning technique I developed was used to answer RQ3.

Prior work in difficult concepts in CS has shown that undergraduate students have difficulties with pointers[43, 78], functions[104, 78], scope[104, 78], loops[43, 78], and control flow[78]. My work looks to use students' expressed thoughts and observed actions to pinpoint the concepts students find difficult, explore the causes of the difficulties, and suggest pedagogical approaches to address these difficulties. I have used various artifacts to qualitatively analyze what students are thinking, drawing from related work to guide which concepts/topics to focus on such as Goldman's Important & Difficult Topics[78]. This related work provides justification for my selection of topics within the CS curriculum, and support for the potential benefits that may result. The three formative studies and the artifacts they collected include: 1) conceptual assessment surveys followed by task-based interviews in which students analyze code and attempt to predict code behavior while explaining their thought processes which collected students' explanations of their reasoning and the audio of task-based interviews as data; 2) "coding in the wild" tasks in which students interact via the edit-save-compile-run cycle of programming to recreate a solution to a simple problem while thinking aloud, which collected screen capture data and the audio files of the think aloud protocol; 3) interview analyses of student design diagrams/documentation in a software engineering course, tasking students to explain their designs and comparing what they believed they had designed with what is actually shown from the documentation, which collected team interviews and the actual design documentation the teams created.

My three formative studies have led to some key insights, including:

1. Students value and improve their work with feedback, whether it is from an instructor, compiler, or interaction and trial-and-error in a development environment.

2. Students attempt to avoid complexity when programming or encountering concepts they do not fully grasp.

3. Students struggle with the planning stages of problem solving.

4. Students develop fragile understanding (having some notion of a concept but not enough to completely solve a problem using it) of some key concepts, and the understandings are reinforced when they have to explain how or why a concept is being used.

Based on the results of these formative studies, I sought to explore what instructors can do to help students overcome these obstacles and to improve student retention of knowledge and retention in the CS major. While gaining a solid understanding of how students are thinking, what misconceptions they hold, and where they struggle, I have managed to identify some problematic concepts and shed light on some of the reasoning students have with respect to those concepts, and leverage this to develop an intervention to improve CS pedagogy.

The idea of feedback and its importance arose in all of the studies and support for the importance of feedback is found in the literature. Higgins[87] found that students would like to receive feedback and even feel that it is owed to them as consumers of knowledge, but also feel that much of the feedback provided is too generic, impersonal, and vague. Tseng[207] looked at the effect of peer feedback on performance and found significant improvements on students' assignments when the students received peer feedback and that students learn from both the process of making improvements to their assignments using peer feedback and also learning from giving feedback on other students' assignments. The MBF intervention that I design employs such feedback and does so in a structured way, based on misconceptions and related work on how novices learn how to program. It employs techniques used in my formative studies, such as the coding in the wild in[105] and the explain your reasoning conceptual assessment surveys in[104]. The technique also draws on related work on active learning techniques such as self explanation, as seen in Chi's work [30]. The technique uses peer feedback, in which one partner (Partner A) is provided with prompts that were developed based on misconceptions and difficulties found in prior work and in related work on how students, particularly novice students, learn to program. The prompts are discussed with Partner B, who is viewing and optionally editing her code.

My final study seeks to evaluate the value of MBF versus human autograder feedback in a quasi-experimental study of students engaged in a CS assignment. With the CS field facing both increasing enrollment in the face of limited resources, and limited representation of minorities, my work seeks to find a strategy to help address both of these realities. I leverage this prior work, introducing and evaluating a pedagogical intervention meant to address misconceptions found in

introductory CS courses. MBF is compared against a human simulated version of autograders, which are becoming a common strategy for addressing the rising enrollment of CS majors[138]. For this study, I evaluated the intervention with 76 students in a second level CS course at a large, public, engineering-focused university in the United States. I use a large, public, engineering focused university to ensure that MBF is feasible at scale, and with the knowledge that active learning techniques have been shown to benefit all, but particularly benefit URGs[82, 73, 126]. Due to population demographics of the university, this study was not able to evaluate the effects of URGs and this would be a goal of my future work. The MBF study consisted of: a conceptual assessment pre-test; attempting to code a short program; an intervention in which peers work in pairs using prompts; and a conceptual assessment post-test. The evaluative MBF study sought to answer the following questions:

RQ1: Can a structured debriefing process based on misconceptions (misconception based feedback) improve student learning outcomes related to a programming assignment?

RQ2: What misconceptions can be reduced using misconception based peer feedback?

RQ3: How do the roles of the students in the structured process (feedback provider/receiver) affect their learning outcomes?

The rest of this document is sectioned as followed:

-Chapter 2 provides the background and related work surrounding my MBF study.

-Chapter 3 provides an overview of the formative studies and their results.

-Chapters 4, 5, and 6 give a detailed description of the formative studies, how they were used, and the results gained from them, using text from either my published or written academic papers.

The order of the studies will be as follows:

Ch. 4: "Explain Your Reasoning" Assessments + Task-Based Interviews

Ch. 5: Coding in the Wild

Ch. 6: Software Engineering Design Documentation Interviews with preliminary analysis

Ch. 7: Misconception-Based Feedback Invervention and Study Design

Ch. 8: Evaluation of Misconception-Based Feedback and Conclusions

Chapter 2

Background and Related Work

The results of my formative studies described in detail in chapters 4-6 and related work suggest: the existence and prevalence of certain misconceptions in undergraduate CS; success in CS requires that misconceptions be addressed; feedback can help address misconceptions; active learning techniques have been shown to help underrepresented groups (URGs) as well as all students. Based on these insights and the related literature, I designed and evaluated an active learning technique, framed in the social constructivist theory and employing structured feedback based on known misconceptions, with the hypothesis that such an intervention could be an effective to address misconceptions as students learn CS topics. In the following subsections, I present related work that informs my approach, motivation, and helps to explain my results.

2.1 Broadening Participation and Rising Enrollments

The CS field currently lacks diversity. Despite initiatives like the National Science Foundation's (NSF) CS4All program[70], data still suggest that underrepresented groups are not pursuing or completing degrees in CS at representative rates[3]. "According to the Integrated Postsecondary Education Data System (IPEDS) data from the National Center for Education Statistics (NCES) (using only codes 11.0101 and 11.0701 in defining CS) for all bachelor's degrees granted in CS in 2015 8.4% were Latinx students at doctoral-granting institutions; 8.5% were Latinx students at non-doctoral-granting institutions; 4.3% were Black students at doctoral-granting institutions; and 8.6% were Black students at non-doctoral-granting institutions[3, 71]." This is compared to the

2012-2015 enrollment rates in college, which were 14.2% for Black students and 16.9% for Hispanic students[175]. The same NCES report stated that women comprise only 17.9% of bachelor's degrees in computing sciences, despite earning 57.3% of bachelor's degrees overall[3]. Another report by the NSF showed that computer, mathematical, and physical sciences had the lowest retention rate (43%) of all science and engineering disciplines when tracking students who enrolled in 4-year institutions in 2003/4 based on 6 year graduation rates[71]. Data from UC Irvine shows computing degree retention rates from 2009 to 2013 as 40% for Black students and 38% Lantinx students compared to 65.8% for White students and 65.8% for Asian students[3]. This report also showed that the Colorado School of Mines had 66.4% retention rates for men and 62.6% for women first-year computing students from 2009-2014, but transfer students retention rates were 78.3% men compared to 54.7% women for this same time frame. These statistics all point toward two issues: that enrollment of URGs in CS is low, even when compared to the overall enrollment rates of the groups in college; that retention rates in CS are low, and this issue is more pronounced for URGs than in the majority population.

The National Academies of Science, Engineering, Medicine, and others[138] found that from the 2012-2013 academic year to the 2014-2015 academic year, CS enrollment more than doubled based on responses from 121 CS-PhD granting institutions (43,391 to 90,604). However, in this same time frame, the number of Black or African American students enrolled in CS only rose 0.5% (4.9% to 5.4%). This is coupled with retention rates being lower for Black students in the CS major as seen in[3]. The National Academies of Science, Engineering, Medicine, and others' report discusses how institutions are considering actions to address enrollment such as placing enrollment caps on the CS major and tightening admission requirements for entry to the major, both of which strategies to address increased enrollment may exacerbate enrollment issues and can have negative impacts on increasing diversity within the field[138]. Tightening admission requirements presents issues such as institutions over-relying on past measures of success (SAT or ACT). Performance on these exams has been found to correlate with socioeconomic factors[76], and underrepresented minorities have reported lower scores, possibly due to disparities in opportunities available to different populations[138]. Placing enrollment caps on the major has been shown to deter underrepresented groups from pursuing the CS major. Many institutions are not considering diversity while trying to address the issue of rising enrollments. Only **46.5%** of doctoral institutions who responded to reports considered potential impacts on diversity to address rising enrollments and **79.8%** reported

not monitoring for diversity effects while transitioning to having higher CS enrollments[138]. This issue begins as early as the K-12 level, with some consideration on enrollment caps looking for students with previous computer science experience. Although this appears to be a reasonable criteria to select students to major in the field, it can have a negative impact on diversity[138]. Universities sometimes test for previous CS knowledge or whether or not students have taken courses such as Advanced Placement Computer Science (AP CS) in high school, which is a course that underrepresented groups do not take as often, with women only representing 22% of AP test takers in 2015 and underrepresented minorities comprising only 13% of AP test takers. There were no Black or African American students who took the AP CS exam in nine states in 2015[85].

To equitably address these needs and issues, researchers have looked into various pedagogical approaches to handle larger enrollments as opposed to policies that have been found to limit enrollment and exacerbate diversity issues. Approaches for addressing rising enrollment and retention include **helpful collaboration**, **giving students a better understanding of CS**, meeting students at their various backgrounds, and **increasing sense of student belonging through positive student-student interactions**[3]. Other work to address these problems, discussed later in this chapter, consists of autograders, active learning techniques, problem solving in CS, and the use of feedback to improve learning are discussed in section 2.4 and all stem from well-known learning theories, discussed in the section 2.2. My approach is a low-resource scaleable pedagogical technique that addresses rising CS enrollments but makes it less necessary to tighten admission requirements and enact policies that have been shown to negatively impact URGs who are trying to enroll. I compare this technique against a version of an autograder, which does also offer benefits to scaling CS courses, but provides more generic feedback that is relatively unable to address misconceptions. My MBF technique focuses on bringing conceptual understanding, helpful collaboration, giving students a better understanding of CS, and increasing sense of student belonging through positive student-student interactions, and increasing success in the introductory CS courses, so that retention rates of students will increase. To provide the conceptual understanding, it is necessary to talk about the theories of learning that support my technique. The next section gives the related work on these theories.

2.2 Theories of Learning and Cognition

The major theories of learning inform the pedagogical approach developed to improve learning outcomes and the literature on cognition and cognitive tools informs my evaluation of the impact of the approach. These are discussed in subsections 2.2.1 and 2.2.2, respectively.

2.2.1 Theories of Learning

Three basic theories of learning have been developed by educational theorists: behaviorism[182], cognitive constructivism[159], and social constructivism[209]. My work relates most to social constructivism, however this section provides background on behaviorism and cognitive constructivism, as they are the predecessors to social constructivism. **Behavorism** is a view of knowledge as a repertoire of behavioral responses to environmental stimuli[182]. Actions are seen as reflexive responses to environmental stimuli or the result of an individual's prior experience with reinforcement or punishment related to such stimuli. Teaching methods based on behaviorism use "skill and drill" exercises to ingrain knowledge and motivate students with positive and negative reinforcement, such as verbal praise or good grades for correct answers and poor grades or bad feedback for incorrect answers. Behavorism was first studied in the late 19th century with researchers dissatisfied with the current research on the human consciousness[206]. Behavior was seen as a more reliable metric to study because it was observable, and the original behaviorists viewed phenomena irrelevant if they had no basis in human or animal behavior. These behaviorists believed that theories needed to be defined entirely with respect to observable data.

Cognitive constructivism views knowledge as actively constructed by learners using mental representations derived from past learning experiences[159]. In this view, learners interpret information differently depending on individual attributes such as their existing knowledge, stage of cognitive development, cultural background, and personal experiences. Cognitive constructivists believe that learning should be thought of as a process of active discovery and that instructors should act as facilitators, providing the necessary resources and guiding learners as they attempt to integrate new knowledge with old and modify the old to accommodate the new. In a cognitive constructive model, students are motivated by intrinsic factors such as major personal investment in the process of learning. Learners are challenged to face the limits of their existing knowledge and accept the need to modify or abandon existing beliefs. Although instructors following the cognitive

constructivist view do use some behaviorist "skill and drill" exercises to help students memorize facts, formulae, and lists, they place greater importance on strategies to aid students in actively taking in new material.

Social constructivism takes cognitive constructivism a step further by emphasizing the role of language and culture in cognitive development[209]. Vygotsky believed that because language is a social phenomenon, human cognitive structures are also socially constructed. He agreed with cognitivists that learners respond to the interpretation of their external stimuli, but argued that learning is also a collaborative process. Social constructivists see the motivation of students as both extrinsic (based on responses to external rewards) and intrinsic (based on factors internal to the individual). With learning considered a social phenomenon, learners are in part motivated by rewards provided by the community they learn within. However, the learner must still have an internal desire to understand and promote the learning process, since she still actively constructs the knowledge. From a teaching perspective, social constructivists promote teamwork skills through collaborative learning methods. The social constructivist approach applies to my MBF intervention, as its basis is allowing students to construct their own knowledge through language and discussion with each other using structured prompts as a guide.

2.2.2 Basic Theories of Cognition

Three main theories of cognition inform my work: Piaget's Theory of Cognitive Development[210], the Information Processing Theory[210], and Vygotsky's Sociocultural Theory[209]. The Theory of Cognitive Development asserts that humans are born with reflexes, which then develop into constructed schemes based on learning and adapting to our environment[94]. Assimilation and accommodation are the two processes humans use to adapt. Assimilation is making our environment fit within our current cognitive structures whereas accommodation is changing our cognitive structures to match our environment. This idea of environment playing a role in the construction of knowledge and cognitive structures helped provide the foundation of constructivism.

The Information Processing Theory examines the interaction between an information-processing system, task environment, and problem space when humans attempt to solve a problem[179]. The information-processing system focuses on serial processing, one input and output at a time, with only 4-7 familiar "chunks" of information being able exist in short term memory at a time. The framework for problem-solving behavior based on the Information Processing Theory is as follows:

1. "A few, and only a few, gross characteristics, of the human information-processing system are invariant over task and problem solver. The information-processing system is an adaptive system, capable of molding its behavior, within wide limits, to the requirements of the task and capable of modifying its behavior substantially over time by learning. Therefore, the basic psychological characteristics of the human information-processing system set broad bounds on possible behavior but do not determine the behavior in detail.
2. These invariant characteristics of the information-processing system are sufficient, however, to determine that it will represent the task environment as a problem space and that the problem solving will take place in a problem space.
3. The structure of the task environment determines the possible structures of the problem space.
4. The structure of the problem space determines the possible programs (strategics) that can be used for problem solving[179]."

The Atkinson-Shiffrin model of the information processing theory asserts that human memory has three components: a sensory register; a short-term store; and a long-term store[4]. The sensory register deals with aspects that directly relate to senses, such as having visual or verbal cues to help remember things. The short-term store retains information from the sensory register, but only lasts for 18-20 seconds after being heard. This short-term store holds chunks of information, usually 7 +/- 2[4]. This short-term, or working memory, can be overloaded if too much information is placed into it. Pieces of information that are transferred from the short-term store end up in the long-term store, which retains the information permanently. These elements of the Information Processing Theory apply to related concepts such as working memory and cognitive load, discussed in more detail in section 2.2.3.

Sociocultural Theory states that a distinctively human characteristic for learning is our voluntary control over our natural biology by using higher-level cultural tools(i.e. language, logic, etc.)[117]. These cultural interactions and processes that we learn and experience throughout life offer us a better way to problem solve novel situations than responding inherently the way we might when we are infants. Through these theories, cognitive tools and other theories have been developed, some of which are useful in explaining observed phenomena. The next section discusses the theories

and tools related to cognition that I have used and have aided in explaining the results in my dissertation work.

2.2.3 Theories and Tools Related to Cognition

To explain and evaluate the results from my formative studies and guide the design of the MBF intervention, I look to several cognitive tools. These tools were chosen for their relevance to the related work on learning that is present in my formative studies, seen in Chapters 4-6. The tools give a framework to explain how we as humans process and interpret information, a key element in understanding what students are thinking and how to address difficulties students have in learning certain CS concepts. I present and discuss cognitive load theory[195], Kahneman's System 1 and System 2[101], abductive reasoning[122], and fragile knowledge[158].

Cognitive load theory (CLT) informs instruction[195] and is based on the idea that there is a limited working memory[6], memory that you can actively use at a given moment, and that working memory is connected to an unlimited long-term memory[109]. The memory elements of CLT relate to the Information Processing Theory[4], discussed above. If a person attempts to utilize too many pieces of information at once, cognitive overload may occur. When it comes to instructional design, the *intrinsic*, or inherent cognitive load of a concept cannot be changed. What instructors can change are the *germane* (relevant), or *extraneous* (unnecessary) cognitive loads. These two deal with how information is presented, with germane load being relevant information for learning and extraneous load being extra information that may not be necessary to learn. Sweller suggests in general that instructors want to increase germane cognitive load and decrease extraneous cognitive load[194]. Worked examples, or problems that provide a step-by-step demonstration of a problem and how it would be solved, have been used to accomplish this in mathematics[196, 41] and computer science[130]. Worked examples for novices are beneficial as they provide organizational structures to store new information and they reduce working memory load and focus student attention directly on relating the necessary elements to solve problems[130, 110]. Kalyuga found that worked examples may be become redundant once learners have some experience with a concept, causing more extraneous cognitive load and not being as effective as attempting to solve problems[102].

When students are presented with and attend to information, it enters their working memory. Many times, the information is presented through consistent lecturing, particularly in some STEM and CS courses where content coverage is a concern. Multiple representations are commonly

used, with graphical and textual representations being difficult to coordinate for novices, as they increase the extraneous cognitive load[40]. Other work has found that mixing auditory and visual presentation modes as opposed to presenting multiple representations individually in a unitary mode can result in a reduction of cognitive load[137]. Hu[92] looked at how humor can have a positive or negative effect on STEM education, depending on whether it is integrated into the intrinsic cognitive load versus added to the extraneous cognitive load of concepts considered already more difficult to comprehend compared to concepts outside of the STEM field. This stream of information coupled with the limited capacity of working memory can lead to issues processing and storing the information being presented. Understanding the implications of CLT on instructional design is important for pedagogical advancements. CLT informs some of the results found in all three of my formative studies in Chapter's 4-6. I next describe a cognitive tool humans use for interpreting information.

Kahneman's **Systems 1 and 2 (S1 and S2)** are a cognitive tool used to explain how we interpret information [101]. System 1 (S1) is responsible for immediate, automatic thinking that solves very familiar problems whereas System 2 (S2) handles problems that require more analysis and calculations. In general, S1 is triggered first and uses heuristics or familiarity to attempt to solve a problem. An example would be most adults being asked the result of 2 + 2. The correct answer of 4 would come to mind automatically and without any real calculations involved (S1). This same question, however, posed to a child who is just beginning to learn about counting and mathematics could be an S2 process. You might notice the child counting on her fingers to get to the accurate answer. Hadar used this tool in CS as a method for analyzing the disconnect between how object oriented (OO) programming works and how the mind thinks it works[83]. He found that even software development professionals with vast experience in OO design and implementation end up exhibiting misconceptions that can be explained by the fast acting and heuristic nature of S1. This cognitive tool aids in explaining some of the results seen in my formative studies, particularly the study described in Chapter 6 such as students using the deep thinking S2 and used strategies such as complexity avoidance to work around the lack of an apparent and intuitive simple solution while designing software engineering diagrams and also offers an interesting lens to look at misconceptions for future work.

Another cognitive phenomenon I have observed students utilize in attempting to reason through programming concepts is **abductive reasoning**. Letovsky describes abductive reasoning in the context of program comprehension as "a plausible inference technique that involves explaining

phenomena by using deductive rules backward to generate possible explanations"[122]. Kovacs and Spens[113] cite researchers defining abductive reasoning as developing new knowledge in a creative and intuitive fashion[2, 108, 199]. One of the famous popular fiction examples of abductive reasoning is what is observed in the work done by Sherlock Holmes[25]. This form of reason also is a common misconception people have, believing that Sherlock Holmes uses deductive reasoning, but in fact, Holmes always generates possibilities that might explain observable phenomena before doing any type of deducing. This creative nature of this reasoning was noted in multiple instances from introductory programming students in my first formative study[104] such as when students seemed to see the name of a function and assume that function did what it was named. For example, assuming a function named "swap" would swap the values of variables without actually fully reading the code to ensure it does swap them.

A common concept that guides the design of my research and helps explain some of the results of my work is students having **fragile knowledge** of the various CS concepts they have learned. Perkins defines fragile knowledge as "the person sort of knows, has some fragments, can make some moves, has a notion, without being able to marshall enough knowledge with sufficient precision to carry a problem through to a clean solution[158]." In terms of computing, Perkins et al. "attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer..."[157]. This fragile knowledge can persist after significant instruction, as Sleeman et al. found when stating "even after a full semester of Pascal, students' knowledge of the conceptual machine underlying Pascal can be very fuzzy[183]."

In 1986, Perkins conducted clinical interviews with students after taking their first year of programming using the BASIC language[158]. In this work, Perkins states his belief that understanding the ways students' knowledge can be fragile in programming can help researchers determine the nature of student difficulties and design improved pedagogy to address these difficulties. This belief guides the work of this dissertation.

In the study, Perkins had students attempt to implement up to 8 short programs of increasing difficulty, all centered around the idea of printing stars "*" using loops. As they worked, an experimenter was there to attempt to help should they run into difficulties. The experimenter would give one of many high-level prompts along the lines of "What does this do?" when students first encountered difficulties. If prompts were not sufficient to guide the students down the right path, the experimenter would then give a hint as to where the student should look. The final type

of assistance was termed a "provide," in which the experimenter would tell the student what code to put at a certain line in order to move the study forward, and would explain why the code was needed.

For this study, Perkins decomposed fragile knowledge into four types: missing, inert, misplaced, and conglomerate[158]. Missing knowledge was considered knowledge that students either did not remember or had never learned. Inert knowledge was knowledge that students could not remember or retrieve during the interview but they did have the knowledge (had learned it), as revealed by clinical probing. Perkins found inert knowledge to occur when learners fail to perform strategic actions that would cause them to link to the necessary knowledge. Misplaced knowledge was when students used coding structures that would be appropriate in some context at a place where it did not belong. Perkins found that misplaced knowledge may occur in situations where a learner has difficulty coming to a reasonable solution, out of a form of desperation. Conglomerate knowledge is seen when students jam together separate, disparate elements of code in an attempt to give the computer all of the information it needed.

Perkins proposed that better cognitive skills should be taught, suggesting the use of elementary problem-solving strategies to improve programming performance and address the issue of fragile knowledge. The results suggested that this self-prompting and simple asking of things such as "What does this do?" can in a number of cases activate the inert fragile knowledge learners have and cause them to correctly implement a solution and helped set the foundation for the development of the misconception-based prompts for my misconception-based feedback study, described in Chapter 7.

This study and its resulting basis for fragile knowledge in the computing field has been cited in multiple other studies looking at misconceptions and difficulties in learning CS concepts: Lister found that novice programmers have a fragile grasp of skills that are pre-requisite for problem-solving[123]; Porter found that using peer instruction, students could understand a question but their understanding was fragile so that they could not apply it to a second question[162]; I found students exhibiting a fragile understanding of pass-by-value semantics[104] and difficult differentiating between pass-by-value and pass-by-reference semantics[105]; and Thompson found students failing to comprehend the complexity and subtleties of cybersecurity solutions[205]. With the knowledge of how students theoretically learn and the cognitive tools used to aid them in understanding concepts (or detract from this understanding and cause misunderstanding) concepts, I now present research

on misconceptions and difficulties students encounter in comprehending certain CS concepts.

2.3 Misconceptions and CS Difficulties

URGs can have false beliefs about the field of CS when first entering the major that can have a negative impact on their success[11]. These negative views can relate to identity and who can/cannot be a computer scientist, and have even been shown to have a negative impact on academic performance[190, 189]. Considering that Kinnunen[107] found motivation and time (persistence) to be the two major categories explaining why students drop out of the CS major, the negative academic impact that CS identity can have is relevant to the retention rates of URGs. In the Kinnunen work, lack of time broke down into several sub-categories: Student prefers to do something else; Student does not plan enough time for the course; **Student underestimates difficulty of parts of the course.** This last sub-category in particular relates to struggling with understanding the content in CS course. The lack of motivation was further divided into sub-categories: No general study motivation; Rewards of studying are not worth what is put into it; **Student underestimates difficulty of course and loses motivation.** URGs having to deal with issues such as stereotype threat[190] and the overly masculine culture of the CS major and courses[29] can lead to retention rates lowering. Despite these identity false beliefs and difficulties URGs students face, research has found that CS grades are not bimodal[151], as in it is not a skill that you either inherently have or don't. Other STEM research found the value in not framing achievement from a deficit mentality, generating examples and providing supporting evidence of Black children excelling in mathematics, despite the issues they may face being underrepresented[121].

With respect to retention rates, research has found that students tend to have higher retention rates if they can get past the introductory years/topics of a STEM field[142, 58]. This has led to a plethora of research on CS misconceptions in the computing education research community, as those misconceptions are the building blocks that lead to students not understanding enough of the course content to persevere through the major[166]. The misconceptions cause the CS content to be more difficult than expected, a major reason for students dropping out of the CS major[107]. A key component of my work centers around misconceptions and using constructivism to address them. Students have been found to form schemas based on common features in their experiences[53]. As no student arrives to class completely devoid of experience, these schemas are founded on students'

existing mental models. Personal experiences shape these mental models, so experts cannot simply replace individual's incorrect models with their own. These misconceptions can be seen on a continuum of novice and expert knowledge systems[185]. As they are rooted from an individual's personal experiences, misconceptions can often persist and resist a "correct approach[185]." Educators are tasked with discovering these misconceptions and creating environments that provide students with feedback that promotes new schema construction and reinforcement[178]. Researchers hypothesize that teachers knowing the common misconceptions their students have as crucial to effective teaching[5]. Sadler particularly found from observing over 9,500 students in 181 STEM classrooms, that teachers who could identify common misconceptions students would have larger classroom gains and increased academic performance by those students[172]. Since research has found value in understanding and being able to address misconceptions, researchers have developed and evaluated various ways to accomplish these goals.

STEM researchers have looked into misconceptions and how they can affect learning in subjects such as biology [112, 84], mathematics[42], computer science[81], and STEM in general[89, 26]. Confrey [38] states that even before formal study, people have "firmly held, descriptive, and explanatory systems" that are different than what's in the curriculum and "are resistant to change through traditional instruction[111, 147]." Tew and Guzdial posit that this may be less true for computer science, and that misconceptions in this domain may be more related to aspects of instruction rather than due to beliefs that students bring with them[201]. Confrey found that within programming, misconceptions stem from systematic errors. Researchers have explored misconceptions on topics including propositional logic[86], memory models and assignment upon declaration[100], algorithms and data structures[152], looping strategies[187], BASIC programming statements [8], language-independent conceptual "bugs" in novice programming[153], and misconceptions and attitudes that interfere with learning to program[33].

A **concept inventory** (CI) is a criterion-referenced test, designed to help determine a student's knowledge of a specific set of concepts, and to expose misconceptions. Taylor et al. have since described the development of CIs for computer science as being "in its infancy[198]." Tew and Guzdial developed the FCS1[201], a validated instrument for CS1, which uses pseudocode in an effort to be applicable across a variety of pedagogies and programming languages. Parker et al. replicated this in an "isomorphic" form as the SCS1 to enable broader distribution and avoid saturation[148]. Decker designed an assessment for CS1 and CS2 in Java[48], which was developed and tested at

her home institution as part of her thesis work. Caceffo describes his process for the development of a concept inventory for introductory programming using the C programming language[23]. The creation of concept inventories relies on the selection of appropriate topics for inclusion. These appropriate topics are generally determined based on common misconceptions students have in CS. Concept inventories serve as a foundation for the "explain your reasoning" (EYR) conceptual assessments I developed in my first formative study, discussed in detail in Chapter 4, and also used as a learning outcomes assessment tool for my MBF evaluative study, seen in Chapter 8.

With a similar goal of comprehending student reasoning when learning to program, McCracken et. al broke down the learning objectives for first year CS students into 1) Abstract the problem from its description; 2) Generate sub-problems; 3) Transform sub-problems into sub-solutions; 4) Re-compose the sub-solutions into a working problem; and 5) Evaluate and iterate. These researchers found that the performance of first year students was significantly worse than expected [132]. McCracken's work gives insight into how students problem solve while programming, which was used for analysis in my second formative study which observed students coding in the wild while providing think aloud data, study seen in Chapter 5.

Other CS education researchers have examined what topics are difficult or error-prone for students, with [43] looking at difficulties with pointers in the C language, [28] using a large dataset of CS1 students solving programming problems to identify challenging concepts, and [91] providing a detailed list of common syntax and logic errors observed in an introductory Java programming class. Cherenkova [28] did this by gathering over 250,000 student responses to weekly code-writing problems with the goal of identifying concepts that students found challenging. Her work showed that students have significant difficulties with conditionals and loops, which persist throughout the duration of a course.

Craig's work looked specifically at the concept of pointers[43]. The data collected and analyzed were a mix of responses to multiple-choice questions and submissions of coding exercises based on pointers. Over 300 students submitted artifacts and results showed that students "confuse an address with a pointer," have trouble understanding the relationship between pointers and arrays, do not have a robust mental model of assignment statements even in their second year, and may apply operators blindly in an attempt to make the types consistent in their code.

When learning to program, students experience other difficulties that are not specifically misconceptions. In comprehending a new program, it is documented that novices and experts behave

differently, with Winslow saying that novices need to learn the facts first before progressing to more complex topics[216, 77]; Detienne stating that novice categories depend on suface structures[49]; and Robins stating that novices are limited to surface and superficially organized knowledge[169]. Some of the issues with novices relate back to the "troublesome language" that is either syntactically or sometimes contextually (the words used are unclear) difficult for students to grasp, as discussed by Meyer[134] and Perkins[156]. Robins et al.[169] suggest that motivation and getting students to gain knowledge related to programming first would be beneficial to make "more effective novices." They also suggest explicitly focusing on programming strategies in introductory programming courses, rather than focusing on the syntax and semantics of programming language features and teaching students how to properly design basic programs to give them the ability to tackle problems regardless of the specific features of the language used. These suggestions are consistent with the observations of Soloway that the major stumbling block for novices learning to program is in "putting the pieces together, composing and coordinating components of a program" rather than in language constructs[186]. Soloway states that planning code implementation and breaking down the mechanisms necessary to enact that plan are essential for novice programmers to learn. Understanding why students struggle and how the misconceptions form when learning CS then lends itself to the important question: What should we do about it?

2.4 Pedagogical Approaches to Address Misconceptions

2.4.1 Autograders

When considering the rising enrollments and how to scale CS courses, the National Academies Press (NAP)[138] calls faculty and professional staff the "most significant resource constraint in CS departments." The authors mention how efforts for routine class management increase with enrollment, and some of these efforts do not easily scale. One suggestion mentioned to address rapid increase in enrollment rates is automated grading of homework, assignments, and tests, providing instant validation of correctness and saving instructors from the amount of time and effort grading can require. Wilcox explored the role of autograding in CS at his institution in Colorado in response to rising enrollments[213]. In his study, a particular benefit was found in the resources that can be saved through automated grading. His analysis found that approximately 288 hours per semester could be saved by automating the grading of assignments and quizzes as compared

to grading them by hand. Based on exit surveys and student evaluations, he even found benefits for academic performance and student interest in the major. Wilcox does mention drawbacks to autograding, such as difficulty replicating code reviews and also how automated grading can have an unforgiving nature. Another key drawback was his result that **some students merely worked on assignments until they would pass the preliminary tests, so they were working to get the scores they wanted as opposed to actually making sure they understood the concepts they were meant to learn.** This issue of failing to focus on a full understanding of concepts in CS is a key issue I wish to address with my work and an important trade-off to observe for the benefits and costs of the resources required for various types of feedback.

Edwards also looked at automatic grading for programs and found benefits[57]. He states that although approaches vary, the general idea behind an automatic grader is that code submissions are compiled and executed against some form of test data provided by an instructor. He cites Curator as a system that had been in use at his institution for some time, and mentions several setbacks:

1. Students **focus on output correctness** first and foremost; all other considerations are a distant second at best (design, commenting, appropriate use of abstraction, testing one's own code, etc.). This is due to the fact that the most immediate feedback students receive is on output correctness, and also that the Curator will assign a score of zero for submissions that do not compile, do not produce output, or do not terminate.

2. Students are **not encouraged or rewarded** for performing testing on their own.

3. In practice, students **do less testing** on their own.

Clemson is currently assessing and considering two autograding tools: GradeScope and Mimir. GradeScope offers the ability to create rubrics quickly while grading paper assignments and has a documented setup for assignment templates and rubrics. GradeScope also provides a series of videos that can be found on their website to help instructors get started with the tool. Mimir is not able to assist with paper assignments, but can be used for grading online quizzes or programming project assignments. There is no additional coding required to set up Mimir to grade students' code submissions.

These autograders have been shown to save on resources and provide some form of feedback that allows students to have performance gains, but even with test cases developed by instructors, autograders generally leave students to wonder why the test cases fail or succeed, not providing

them the tools to address problematic concepts. Particularly for underrepresented groups, these autograders do not allow the benefits of alternative learning environments in classroom[180] or collaborative problem solving and interdisciplinary projects[154, 164], which have been shown to improve diversity[138]. Autograders address a problem, but do not currently focus on shifting pedagogy, instead offering a convenient and quick way to reduce workload on instructors, leaving it still in their hands to choose to improve pedagogy in a way that benefits all students. The autograders are more focused on scale, and although they can provide mass feedback, as Wilcox noted, they cannot replicate code reviews, which have been shown to be useful[213] and they run into the issue of students attempting to code until they get the scores they desires as opposed to working to learn and understand the concepts[213]. These graders also provide mass feedback, leading to feedback that is more generic, an issue that Higgins noticed students wanted to avoid when receiving feedback[87].

I look specifically at this feedback element of autograders, as although it does offer benefits to scaling CS courses, the more generic nature of the feedback and the relative inability to address misconceptions can negatively impact diversity. Without addressing these misconceptions, students, particularly underrepresented groups, can struggle with course content, which often leads to the students not staying in the major, retention issues discussed in the report to the President to increase STEM degrees[142]. My work seeks to offer a method that will allow CS courses to continue to scale and be less resource-intensive than with all grading and feedback being performed by hand but also provide feedback to students with significant benefit to the learning of concepts and to addressing misconceptions.

Some autograders do attempt to provide more than confirmation feedback, as seen in Cao et al.[24]. This study looked at a version of autograder (GradeScope) that provided feedback based on a rubric, which students preferred to exams graded by tests. GradeScope offered instructors the opportunity to provide a defined rubric that gave students more detailed feedback on CS exam questions that were not multiple-choice, and the rubric could be adjusted as instructors graded exams. The software also allowed students to submit grade resubmission requests online, which many students took advantage of because of its convenience. However, the study found no significant time difference between online and paper grading when both prep and grading time were taken into consideration. The result suggests that in their current form, autograders that can provide more substantive feedback do not save resources as well as those that simply provide confirmation

feedback. It is important to note that the generation of a rubric and selection of appropriate feedback for a particular response is supported and streamlined by this tool, but does not constitute a fully automated system.

Quality feedback may have one of the most significant impacts on student learning[10, 72]. Boud defined feedback as "the process whereby learners obtain information about their work in order to appreciate the similarities and differences between the appropriate standards for any given work, and the qualities of the work itself in order to generate improved work"[13]. Most automatic grading tools found in a review done by [106] would not provide useful feedback as defined by Boud.

The work that is the focus of this dissertation takes these elements of feedback (described further in 2.4.4) and self-explanation (described further in 2.4.2) and applies them to an intervention meant to improve the understanding of CS students through the use of guided feedback. The following subsections dig deeper into the related work that connects my research, with a particular focus on my misconception-based feedback study, described in Chapter 7.

I take a constructivist stance, asserting that people try to make sense of the environments that they engage with while learning novel content. This process continuously happens and knowledge frameworks are refined as new environments arise[53]. This view places teachers not as the keepers of knowledge and the only way to gain it, but instead, as guides and facilitators providing students the tools, instruction, and assistance to learn through their own active discovery of knowledge. To this end, active learning offers a valuable pedagogical approach to addressing misconceptions.

2.4.2 Active Learning

From the two constructivist theories (cognitive and social), researchers in education have developed a pedagogical approach called **active learning**[19], which involves students in their own learning process, as opposed to having them merely listen to traditional lectures. Compared to approaches that derive from behaviorist theories, active learning techniques attempt to build on the previous knowledge that the learner possesses, employ discovery learning[19], identify and remedy misconceptions, and often employ collaborative paired or group work supported by social constructivism[7, 209].

Research has shown that active learning techniques are beneficial in various fields. When looking at STEM in particular, Freeman performed a meta-analysis of the literature that found 6% increases in exam scores, learning improvements in the form of conceptual gains, and lower

failure rates in courses using active learning (21.8% versus 33.8%)[73]. For underrepresented groups (URGs) in STEM, active learning has been shown to help bridge the gap between these groups and the general population. For example, Haak et al. conducted a study looking at the performance of over 100,000 students who took University of Washington biology courses from 2003 to 2008[82]. The study compared the grades of regular students with students in the school's Educational Opportunity Program (EOP). These students had been shown to historically perform worse at the university and come from educationally and economically disadvantaged backgrounds. Of the EOP students in this study, 76.5% were also considered an underrepresented minority based on race. The results of this study showed that when the course was taught using a highly structured, lecture-free format that incorporated active learning techniques such as pre-class reading quizzes, extensive informal group work, peer instruction[44], and weekly practice exams, not only did all students significantly improve their grades, but students from the EOP group had a disproportionate benefit to students not from the URG[82]. This benefit was statistically significant, even when controlling for instructor by comparing sections taught by the same instructor but in different ways (lecture-based vs. structured and lecture-free active learning).

In a different study, Lorenzo et al. found benefits for active learning in reducing the gender achievement gap for students in a Physics course[126]. This study collected data from 6 years of a physics course taught at Harvard that had an average enrollment of 202 and with a 1.7 to 1 male to female ratio. The study compared traditional lecturing instruction style with peer instruction (PI) style[44] and found that when using the active learning technique of PI, female students improved significantly in the interactive engagement courses[126].

Even though we know that student attention is necessary for learning, research shows that student attention falls off after about 10 minutes in a standard lecture[12, 193]. Having intermittent active learning activities that offer breaks from the standard lecture offer a solution to the attention problem and have been shown to increase attention at the time of the activities[135].

Research shows that such *retrieval practice*[103], also known as *test-enhanced learning*[165], is associated with improved retention and transferability. These benefits are very useful in content-heavy fields, allowing students to fully understand concepts as opposed to having fragile knowledge (having some notion of a concept but not enough to completely solve a problem using it) associated with the concepts they are learning. Other key factors to improving the benefits of active learning techniques are ensuring the techniques are challenging enough to evoke more critical think-

ing, spreading them out over time to allow the information time to be digested, and providing quick feedback to students so they can understand what they are doing and where they can make improvements[17, 170].

A number of factors support the benefits observed with active learning, including reduced extraneous cognitive load on working memory by engaging students[194], increasing students' retrieval practice (strategy to bring information to mind and meant to enhance learning)[103], and increased attention[135]. A benefit of active learning is that it allows students to practice the new information that they are obtaining, relate it to knowledge and experiences they already have (constructivism), and gives their memory a chance to rehearse the concepts, reducing cognitive load. This can lead to the knowledge being encoded in long-term memory as opposed to working memory[63].

Within CS, researchers have studied multiple active learning technique that instructors have employed. One of the more common techniques is pair programming, is defined as "a practice in which two programmers work collaboratively at one computer on the same design, algorithm, code, or test"[214]. Pair programming, which engages students to co-construct knowledge, collaboratively searching for regularities and testing their code and aiding in forming schemas[20, 45], has shown that both driver and navigator contribute information[20]. Williams also found that the pressure from having another person observing and helping you work, "pair pressure," helps keep students focused[214]. Other benefits to this technique are that it has been shown to produce greater enjoyment and faster completion times of assignments[36]. Although pair programming has been used in many contexts and offers learning benefits, the context and nature of the discussion occurring during this technique is important.

Clark found that discussion needs to be *grounded* in points of mutual understanding to operate productively in a joint problem space[34]. To coordinate and facilitate student thought processes, *task-specific support structures* can be used or scaffolding that helps students model their domain knowledge[45]. My work also leverages the common practice of *rubber ducky debugging*, which involves talking out loud to an object, such as a rubber duck (or person) that does not understand the program[96]. This process requires a programmer to slow down and make more deliberate and planned decisions while attempting to code[32]. Research has shown this attention to design and doing code reviews effectively streamline the development process, but students and professional developers strongly avoid them. Active learning activities such as pair programming engage both the driver and navigator in a continuous design and review process, ideally in an encouraging

environment[215]. This helps to mitigate common concerns relating to instructor feedback, such as it being both time consuming to produce and students finding it too generic or impersonal[87].

Self explaining study

A study not conducted in the CS field but that relates to and forms the basis of rubber ducky debugging is Chi's self explanation study. Chi looked at the benefits and effects of self explaining, which is an active learning technique requiring students to attempt to explain to themselves what they are trying to learn[30]. The study attempted to show that a learning technique can be applied and show benefits independent of a particular domain, so it used two courses, biology and physics. The biology intervention looked at declarative knowledge whereas the physics intervention looked at procedural knowledge. The biology intervention used 8th graders as participants whereas physics used college students. It was shown that a statistically significant correlation exists between the number of self explanations and self explanation inferences (parts of the self explanation students are inferring and are not directly stated in the information that they are learning) and post-test scores, with the students all starting from approximately a similar point in terms of prior knowledge coming into the study.

This MBF intervention uses an augmented version of self explanation, employing a peer dialogue with a structured guide of prompts to encourage learners to address misconceptions in a manner that promotes learning. Theories such as constructivism, allowing the students to construct their own knowledge based on their own experiences and environments, help support the benefits of self explanation. Social constructivism is particularly relevant, as the technique requires peers to engage in a dialogue with each other. By incorporating aspects of Chi's self explanation, pair programming, and rubber ducky debugging through structured review prompts based on misconceptions, I provide an environment that promotes "higher order thinking that involves active control over the cognitive processes engaged in learning," or *metacognition*[125] and productive quality feedback loops for students to collaborate and build knowledge. Metacognition is an important factor in planning and problem solving, allowing us to not just decide how to approach a learning task, but also keep track of our comprehension and evaluate progress towards successfully completing this task. McCracken's framework[132] of problem solving in CS, discussed in the following section, centers around these metacognitive principles.

2.4.3 CS Problem Solving

Trial and Error Trial and error is a programming technique that focuses on testing and editing code until correct output is finally produced[58]. It emphasizes the applying and creating phases of Bloom's cognitive taxonomy[1], described in more detail in Chapter 7, which are commonly focused on when teaching undergraduate CS courses[21, 22]. Although the approach can be useful for beginners learning to program, it can also cause conceptual problems as students are less equipped to use the analyzing and comprehending phases of Bloom's cognitive taxonomy, instead focusing on the creating phase[21, 21, 58]. To address this problem solving strategy, Buck and Stucki discuss an "inside/out" methodology when learning to program[21, 22], focusing on designing code writing tasks that are more constrained and target the comprehension and analysis levels of Bloom's taxonomy that the trial and error approach usually fail to capture. The design of the structured prompts, seen in Chapter 7, use this related work to assist students in comprehending, analyzing, and evaluating their programming and problem solving skills, and applying this knowledge to create working code.

Osborn's Creative Problem Solving

Alex Osborn developed a process for solving creative process[146]. The original version of this process consisted of seven stages:[98]

1. **Orientation:** Pointing up the problem.
2. **Preparation:** Gathering pertinent data.
3. **Analysis:** Breaking down the relevant material.
4. **Hypothesis:** Piling up alternatives by way of ideas.
5. Letting up to invite illumination.
6. **Synthesis:** Putting the pieces together.
7. Judging the resultant ideas.

This model was eventually revised to become the Osborn-Parnes creative problem solving process, which consisted of the following five stages: **Clarify**, **Idea**, **Develop**, and **Implement**. During **clarify**, programmers analyze information, goals, and challenges. They then use divergent thinking (exploring many possible solutions) in **Idea** and convergent thinking (narrowing solutions down)

in **Develop**. Finally, **Implement** is when those solutions are applied and tested. This structured process has been used to teach programming in computer science education[158, 131]. For my MBF intervention, the structured prompts are designed to have students focus on the **Idea** phase and then evaluating those ideas through discussion, which falls more in line with McCracken's[132] work.

McCracken Framework

McCracken et al. worked on a multi-national, multi-institutional study of 216 students to assess programming skills of first year students[132]. Using the Association for Computing Machinery's 2001 Computing Curriculum[202] and the computing education expertise of the research team, McCracken et al. developed a framework for learning objectives of first year programming students. This framework is derived from the universal expectation of CS instructors that students learn to solve problems in order to create code that compiles, executes, is correct, and is in the appropriate form. McCracken's framework consists of the following five steps:

1. **Abstract the problem from its description:** This first step involves students taken a problem or program specification and determining the relevant pieces. Once those pieces are determined, they should then be modeled abstractly in the appropriate manner (i.e. pseudocode, models/chart, etc.).

2. **Generate sub-problems:** This step involves breaking down the problem into smaller problems. It can consist of looking at the various functions or classes associated with the programming problem, variables and types necessary, or other elements such as control flow.

3. **Transform sub-problems into sub-solutions:** This step is when students choose the implementation strategies for the sub-problems determined in the previous step. This step can and should also include testing these strategies.

4. **Re-compose the sub-solutions into a working program:** This step involves taking the sub-solutions and combining them to generate a full solution to the original problem.

5. **Evaluate and iterate:** This final step is determining whether the student has solved the problem and if not, making the appropriate changes. This process is iteratively done until a proper solution exists.

McCracken's framework was designed by CS experts and educators to focus on problem solving from the perspective of programming and has had widespread usage in the computing education research

community. It is specific to the CS context and relevant to my work on learning how to program. For my MBF intervention, this framework was relevant in the development of the structured prompts, as described in Chapter 7.

Soloway

Another well-established method of problem solving in the context of programming and CS was found by Soloway. Soloway researched how to revise CS curriculum to teach problem solving skills in the context of learning how to program[186]. He states that "the focus on instruction of the syntax and semantics of programming language constructs leads to an emphasis on the program as the output of the programming process." Soloway broke down programs into two audiences: the *computer,* which is a mechanism that takes instructions from a program that tell it how a problem can be solved, and the *human reader,* who needs to be able to explain why the program is solved. This view then claimed that learning to program is not just about constructing the mechanisms, but also constructing explanations on why the mechanisms work. This approach to problem solving is modeled in the design of structured prompts for the MBF technique, as students do not simply have to describe the mechanisms used in their program, but also reflect on and discuss/explain why those mechanisms work.

2.4.4 Feedback

Higgins [87] performed a longitudinal study looking at the role feedback has in assignments. This was in business and social science units, but there seemed to be a consensus that students preferred and even felt as if they, as paying consumers, deserved feedback, and the feedback that they wanted was not strictly grade-focused. Many of the students genuinely wanted to learn and felt that some of the feedback they received was not particularly helpful with that process. Some feedback was found to be too generic, sometimes it was just hard to read (this study examined handwritten feedback), and other times the advice was vague with no guidance as to how to actually solve the problems identified in the feedback. Their survey found that most students do read feedback, but that the majority spend less than 15 minutes in doing so. Researchers posit that some instructors do not write involved feedback because they do not believe their students are reading it, which leads to a continued problem of communication between student and instructor. Another preference of many students was that feedback be more personal and not as generic[87]. A goal of my work is to ensure that faculty time is being effectively utilized given rising CS enrollments, as some instructors

feel the need to provide detailed and personalized feedback for all assignments, but this may become less feasible as more students enroll in a course.

Tseng conducted a study examining online peer feedback in a high school course learning about computers[207]. The study had 10 peers anonymously give both qualitative feedback and quantitative scores to an assignment three times over the course of six weeks. The assignment required the students to use search engines to develop an itinerary for some activity. The students then made adjustments and improvements to the assignments based on this feedback. A statistically significant increase in the scores was found between the first submission and the second and between the second submission and the third. Two experts also gave scores for the assignments at three feedback points, but students never saw the experts scores to ensure they were not just making improvements based on trusting the authority of the expert instructors. Of note, a significant increase was also seen in the experts' scores of the assignments, even though the experts did not have access to the feedback the peers were giving the students. The scores of the peer evaluations were significantly highly correlated with those of the experts, suggesting that peer assessment can be used as a valid assessment measure for assignments. Tseng states that "the learning in the peer assessment process comes from both students' adaptation of peers' feedback and their assessment of peers' projects." That is, students are learning from both the process of making improvements to their assignments using peer feedback and from giving feedback on other students' assignments. This dual nature of learning by analyzing as well as responding and adapting to feedback help guide the design of my final study. It ties back to the social elements that are an integral part of the social constructivist theory of learning. Tseng also claims to "believe in the importance of peer feedback for the peer assessment and assert that in the process of peer assessment, the students continuously gain formative feedback from peers; therefore, we could observe that the students in the present study improved their projects," tying back to Higgins' [87] results and conclusions about the value of formative feedback.

Le's work classifies adaptive feedback in educational systems for programming[120]. He identifies five types of feedback supported by programming: Yes/No feedback, syntax feedback, semantic feedback, layout feedback, and quality feedback. Yes/No feedback, or confirmation, merely gives a response as to if something is done correctly. This would apply for automatic graders that are testing code execution against provided test cases. Syntax feedback is based on what the compiler provides the user. This would be standard error or warning messages that compilers issue when

attempting to compile code. Semantic feedback focuses on errors in code that stop the program from fulfilling a required task. This feedback can be broken down into two types: Intention-based, which gives feedback based on what the programmer meant to do; and code-based, which merely provides feedback about code that is not semantically correct. Layout feedback relates to the style of code and provides information about how it should be arranged and formatted. Quality feedback focuses on how efficient the code is with respect to time and memory use. It also provides the confirmation feedback of if the code works. Of these, syntax, layout, and quality are specific to the domain of programming.

My focus is on intention-based semantic feedback and allowing students to address, explain, and justify their intentions to remedy misconceptions. Gusukuma[80] attempted to develop a system to provide syntax feedback that is based on misconceptions as opposed to the standard compiler feedback. My MBF study builds on this idea of using misconceptions as a framework to provide feedback, designing prompts based on observed misconceptions from my formative studies, these prompts requiring students to engage in self-explaining and peer dialogue to justify how they decided on their design and implementation.

Conceptual engagement

A successful pedagogical approach requires that conceptual change occurs, and learners are not merely regurgitating what they were instructed on. In science education, Posner identified four conditions necessary for the conceptual change process [163]. When knowledge structures are firmly entrenched, they are highly resistant to radical change and so the first requirement is that the learner must be dissatisfied with the original knowledge structure. The second is that new knowledge structures must be intelligible and make sense. The third is that the new knowledge structures must seem plausible to the scientist or student attempting change. The fourth is that new conceptions must be "open to areas of inquiry," or be able to lead to new insights and hypotheses. Dole re-conceptualized this model and looked at elements such as: the learner's strength, coherence, and commitment to the existing concept; motivation (dissatisfaction, personal relevance, social context) towards an existing concept; whether the message is comprehensible, plausible, coherent, and rhetorically compelling; and how engaged the learner is with a new concept as to if conceptual change will occur [52].

Forman et al.[69] mentions teacher-student authority and how Tabak and Baumgartner [197] describe three roles: monitor, mentor, and partner. The monitor merely observes what the student is doing. The mentor places the teacher as the highest intellectual authority and generally has the

teacher doing most of the speaking with the students listening. Shifting to partner allows for more discourse, and this model helps to frame my hypothesis that the gains will be significant even when using a peer as the person providing feedback and giving each other a chance to reflect of their decisions. My final study emphasizes this partner role and removes the element of authority, having the partner serve as a peer.

Immediate Feedback Assessment Technique

Another technique that incorporates feedback is Epstein et al.'s work on the Immediate Feedback Assessment Technique (IF AT)[59]. The IF AT was designed in a way that the correct choice would have a star underneath it and would be seen as students scratched off that choice. If the incorrect answer was chosen, a blank space would be seen underneath the letter choice. Students were instructed to answer questions normally, but if the answer was incorrect, to rethink their strategies and attempt to answer again. The IF AT allowed students to answer as many times as necessary until they answered correctly. Epstein et al. conducted multiple studies to compare standard Scantron multiple-choice assessments to their Immediate Feedback Assessment Technique (IF AT) version of multiple-choice assessments[59]. These studies found that students had significantly more retention of material when receiving immediate feedback as compared to the no feedback provided by standard multiple-choice assessments. In fact, it was stated that a multiple-choice examination that "does not employ feedback may promote misconceptions"[59]. The implication of these results suggest that feedback is a necessary element of students learning and not gaining more misconceptions.

Although feedback is positive, it is important to provide students with quality feedback. Nicol et al. defined seven principles of good feedback practice[140]:

1. **helps clarify what good performance is (goals, criteria, expected standards);**
2. **facilitates the development of self-assessment (reflection) in learning;**
3. delivers high quality information to students about their learning;
4. **encourages** teacher and **peer dialogue around learning;**
5. encourages positive motivational beliefs and self-esteem;
6. **provides opportunities to close the gap between current and desired performance;**
7. **provides information to teachers that can be used to help shape teaching.**

The bolded items are addressed with my misconception-based feedback technique. To ensure that students were clear on what good performance was, I provided both written instructions for the coding task and also an exemplar, which are found to complement and help strengthen the clarity of written instructions[145]. The facilitation of self-assessment occurs through reflection, which is a key activity in which the MBF prompts require learners to participate in. Students were tasked to assess code they had already written based on prompts provided. With respect to encouraging peer dialogue around learning, the technique is designed explicitly to encourage and facilitate peer dialogue around learning. In order to provide opportunities to close the gap between current and desired performance, the technique allows students to edit their code while going through the misconception-based feedback technique.

My MBF intervention, described in Chapter 7, takes these elements of feedback and conceptual change, and applies them to an intervention meant to improve the understanding of CS students through the use of guided feedback and then Chapter 8 describes the study that I used to evaluate this intervention. The feedback prompts were developed based on the difficulties I found students had in CS topics as seen in my formative studies and related work on how novices learn to program. The prompts allow students the opportunity to learn through explaining their thought processes and having to discuss them either with a peer while observing code that had previously been written.

Overall, this work is motivated by the desire to help address low retention rates in CS, a major where enrollments continue to rise. Research has shown and I have personally observed how URGs can be affected more than majority students by these low retention rates due to issues such as false beliefs about CS identity and stereotype threat, which lead to academic underperforming. As both lack of motivation and time due to difficulty of content have been shown to be key reasons for CS students not staying in the major, I have worked to address the difficulties students have with CS content. The MBF intervention was developed to address difficulties and misconceptions in introductory CS courses, because these courses have been shown to be pivotal in helping to retain students in the major. Research in active learning supports its benefits for all students, but has shown that it can be particularly useful for URGs. Since feedback has proven valuable in addressing misconceptions in related work and my own formative studies, the MBF intervention focuses on quality feedback to improve conceptual understanding for important and difficult topics in CS. In this dissertation, I pilot this technique at an institution with resources to focus on using the technique

at scale, and hope to in the future to test the effectiveness of it with URGs in the context of minority serving institutions. The next chapter gives an overview of the formative studies I conducted and their results that led to the development and evaluation of the MBF intervention.

Chapter 3

Overview of Formative Studies

Before designing and evaluating the MBF intervention described in this dissertation, I engaged in a series of formative assessments designed to identify topics in the undergraduate computer science (CS) curriculum that students find difficult and to characterize the nature of these difficulties. Two of these studies have been published in well-established, peer-reviewed conferences in computing education research and a third has been conducted and preliminary analysis helped to form the development of my final evaluative study. Chapters 4 and 5 contain the published papers associated with the first two studies, and Chapter 6 provides a detailed description of the third study and the outcomes of the preliminary analysis. This chapter contains an overview of these studies and their findings.

3.1 Explain Your Reasoning Surveys + Task-Based Interviews

My first formative study involved development of a survey meant to gauge how students think about selected introductory programming concepts. I pulled from Goldman's related work on important and difficult topics in CS[78] and intersected topics found there with local context of what Clemson teaches in our introductory programming course. This local context was used as Clemson was the location for the study, so I needed to ensure that the topics I explored would actually be covered in the course I was using for the study. Of the topics identified, the focus was on the five

defined below:

1. Parameters/Arguments (PAI/II/III):
   (a) Call by Reference v. Call by Value: Understanding the difference between "Call by Reference" and "Call by Value" semantics
   (b) Formal v. Actual Parameters: Understanding the difference between "Formal Parameters" and "Actual Parameters"
   (c) **Parameter scope, and use in design**: Understanding the scope of parameters, correctly using parameters in procedure design
2. **Procedures/Functions/Methods (PROC)**: (e.g., designing and declaring procedures, choosing parameters and return values, properly invoking procedures)
3. **Scope (SCO)**: (e.g., understanding the difference between local and global variables and knowing when to choose which type, knowing declaration must occur before usage, masking, implicit targets)
4. Assignment Statements (AS): (e.g., assigning values from the right hand side of the operator to the left hand side of the operator, understanding the difference between assignment and a mathematical statement of equality)
5. Control Flow (CF): Correctly tracing code through a given model of execution

From these topics, 21 questions were developed, each designed to focus on a single CS concept. Each question asked about the behavior of a "snippet" of code. Some questions were multiple choice, with only one "distracter," (incorrect answer meant to reveal a common misconception), and some questions asked students to provide the output of the code snippet. In all of the questions, students were asked to "Explain your reasoning," (EYR) and given substantial space to write the thought process behind their answers. This was administered to 106 students in a CS2 (2nd-level CS course) at the beginning of the semester. The surveys were collected, transcribed, and then analyzed as described in Chapter 4. Following this process, follow-up task-based interviews (TBIs) were given to 10 students who had also completed CS1 but had not taken the EYR surveys. These were conducted one-on-one and lasted 60 minutes, with students answering 8 questions of interest that were a subset of the original 21. Students were recorded and tasked with explaining the code

snippets and I was able to probe deeper into explanations that did not seem clear. The four questions of interest that emerged based on the extent to which students struggled and that provided the most information about misconceptions were Pass by value vs. Pass by reference (Q11, Q14, Q17), Scope (Q14, Q17, Q18), return values (Q14 + Q17).

Q11, found in section 4.1, includes a void function **swap** that swaps two values. The **swap** function exchanges the parameters x and y via a local *temp* variable. The function contains a return statement but is void and no value is returned. The **main** function initializes two variables, *cat* and *dog* to the values 5 and 8, respectively, and then invokes the **swap** function, passing in the values of *cat* and *dog* as parameters. However, since the parameters are passed by value, the value of *cat* never actually changes. Students are asked what the value of the variable *cat* will be after the function returns. They are given two choices: 5 (correct), in which the value of *cat* does not change and 8 (incorrect), which would result if pass by reference semantics were in effect rather than pass by value semantics.

Q14 found in section 4.2, involves a void function, **findArea**, which defines a local variable *area* and then assigns to *area* the product of the *length* and *width* parameters. In the **main** function, a separate variable *area* is defined and initialized to 0 and variables x and y are defined and initialized to 4 and 8, respectively. **findArea** is then called, passing x and y as parameters. As in question 11, the function is void and no value is returned. In **main**, a print statement utilizes the *area* variable. Students are asked what the result of the execution will be. The correct answer is that "The area of the shape is 0" should display.

Q17, found in section 4.3, is nearly identical to Question 14 and provides a consistency check across questions. The function **subtract** accepts two integers as parameters, computes the difference and assigns the result to local variable *answer*. The **main** function also declares a variable *answer* and initializes it to the value 7. A print statement is called using the *answer* variable within the scope of **main**. The correct answer is b, since the *answer* variable that exists within **subtract** is a different variable than the *answer* variable in **main**.

Q18, found in section 4.4, introduces a global variable, *answer*, initialized to the value 7 in line 1. The function **subtract** takes in two integers and returns an integer. Function **subtract** initializes a local variable *answer*, sets it equal to the difference between parameters x and y and returns the local *answer* to the **main** function, where it is assigned to the local variable *solution*.

Line 14 has a print statement that prints *answer*, with the intended *answer* being the global variable, as the *answer* variable from the **subtract** function is out of scope. The answer choices provided allow students to either choose the result of **subtract** (5-8), or the value of the global variable *answer*, which is 7.

Below are some key findings based on four questions of interest of the 21 questions of the survey.

3.1.1 Pass By Value

In the EYR study when students' responses indicated misconceptions related to the idea that invocations of functions using pass by value pass a *copy* of each argument and the function modifies only this copy and not the original value in the calling code. Evidence of this misconception was seen in Q11, where students gave responses such as "swap function simply swaps 2 values using a temp variable to hold one value in memory while it is replaced by the other" or "I saw that it was a swap function at the top, so I assumed it did what its name was. After that, I solved the main, checking to see if y'all might have tried to trick us by reading the swap function. Then I looked at cat, and picked the number it wasn't equal to." This finding confirms prior work by others that introductory programming students struggling with pass by value semantics Goldman et al.[78], and others[33, 129, 65, 27, 100].

One explanation for this misconception in the context of the EYR study is that students engage in abductive reasoning, working backwards from seeing the name of the function "swap" and making sense of the code by trusting or assuming that the function will perform the action its name suggests. Another explanation is fragile knowledge, with many students having only recently learned about using a "temp" variable to swap values and so their attention is focused towards this newly acquired knowledge but they are not able to also attend to the fact that the function is pass by value because they are trying to process too much information (i.e. cognitive overload). Fleury's work found that students constructed their own rules about how parameters are passed, which is fine in some cases, but can lead to incorrect mental models as their rules are not the actual way the concept works[65].

This misconception seems to persist and has been observed in multiple studies and contexts, so my work would suggest that instructors consider alternative ways to present the pass by value concept, such as beginning with a plan-like approach to build understanding with visual, concrete

examples. This approach can be supplemented with "puzzler" questions similar to those in the the EYR survey to allow students the opportunity to form a detailed understanding of the semantics of language features such as parameter passing. Another suggestion to address Fluery's results of students constructing their own rules would be to have students engage in activities such as think-pair-share[63] and self-explain[30] to prompt students to directly address how they are conceptualizing parameter passing. Then, if students exhibit incorrect mental models, they can be dealt with earlier on in the learning process.

3.1.2 Overwrite "False-sharing"

The EYR study also revealed a misconception in which students believe that a variable with the same name but in a different scope can be modified by a write to the local variable. This was seen in Q14 of the EYR study, where students gave responses like "The area of the shape is 32. The main function passes the values of x and y to the function findArea. findArea is called with the input parameters of x and y, which = 4 and 8, respectively. Area = 4 * 8 because it is written in the findArea function" and "32. The integer area is returned but I think the compiler won't like that area is being defined at 2 separate times during execution (not sure how that will impact the program)." This misconception has also been seen in related work. This idea of overwriting a local variable is a more specific version of a scope misconception. Scope misconceptions were found in Goldman's work identifying important and difficult programming concepts[78] and memory model misconceptions have been reported by Kaczmarcyk[100].

We saw that some students think that variables with the same name but different scopes are actually the same variable. This, similar to pass by value, could be the result of students engaging in abductive reasoning, foraging for a reasonable explanation of code that does not operate in an expected manner and attempting to make the most sense out of it. It could also be that students do not have a strong mental model of how the concept of variable names and memory locations work.

Knowing that students have this misconception, instructors would want to to start with straightforward applications of plan-based instruction and then move on to "puzzler" examples such as the one in Q14, requiring students to walk through and explain what is occurring in both situations to ensure that they build correct mental models about memory and scope. Another suggestion would be to have students engage in activities such as think-pair-share[63] and self-explain[30] to prompt students to directly address how they are conceptualizing memory and variable scope so that if there

are incorrect mental models, they can be dealt with earlier on in the learning process.

3.1.3 Global Access

The EYR study also showed that students had the misconception that global variables, which are variables visible in all other functions and blocks unless a local variable of the same name hides them, may not be accessed from the main function. For example, in Q18 in the EYR study, students provided responses such as "Because answer = 7 isn't in the main scope" and "Line 1 has no affect on answer." This finding is novel. This misconception has not been identified as a common misconception that introductory programming students encounter and it was unexpected when we uncovered it.

An explanation for the existence of this misconception is that the context in which global variables have been taught is generally through the use of a global constant, and so students did not have a full grasp of the concepts of how global variables operate. Student explanations on this topic suggest that students learn in a plan-based versus syntax-based way. Instructors must ensure that students experience a sufficient variety of use cases to fully demonstrate the desired concepts or functionality.

3.1.4 Global Overwrite

In the EYR study, we observed students with the misconception that a global variable can be modified by writing to a local variable of the same name. This is a more specific type of the OW misconception. Evidence of this misconception was seen in Q18 of the EYR study, where students provided responses like "because answer is a global variable, the subtract function will change the value" and "Answer was not a const, so after 5-8, answer is returned as -3. Since the global variable answer is not a constant, the value of it can be modified, and it is just before the print statement." In task-based interviews (TBIs), students provided further evidence with the response "this gets reinitialized up at the top from 7 back to -3...Because that's a global variable but it's not immutable. I mean, it can be changed." This finding is not novel. As mentioned at the beginning of this section, the GlOW misconception is a specific instance of the OW misconception, which itself is a form of a scope misconception. Scope difficulties were found by Goldman in his work[78] and memory model misconceptions have been reported by Kaczmarcyk[100].

Lack of ability to transfer knowledge between domains as well as fragile knowledge help explain this misconception, as students seemed to grasp what global variables are, but are not familiar with using them outside of the realm of their use as constants, where they are not intended to be modified. This misconception coupled with the OW misconception suggest that there is a lack of understanding of *variable shadowing* (thinking that variables of the same name are the same variable) and a lack of knowledge of how return values are passed back to calling code. Based on Clemson's introductory programming course structure, students' exposure to variable shadowing is limited and it seems they possess fragile knowledge about the construct.

To address this issue, instructors may wish to expose students to situations in which they experience unexpected outputs when they attempt to associate uses of variables with definitions that occur in different scopes. Being aware of this misconception offers instructors the opportunity to design appropriate in-class examples or to incorporate associated elements into projects that would help students have a more solid understanding of scope and challenging, uncommon situations such as variable shadowing. Such active learning techniques, if conducted in pairs or small groups, could leverage the benefits suggested by social constructivist theory and promote student formation of appropriate mental models associated with scope and visibility of variables.

3.2 Coding in the Wild

My second study was designed to gain insight into what students are thinking by observing what they are actually doing. I wrote a simple menu-based calculator program that asks a user to select a function (addition, subtraction, or flipping the signs (changing positive to negative and negative to positive). It then prompts for two integers and performs the selected operation on those two integers. The program executes in a loop, so it can be repeated until the user enters the exit code (any number other than "1"), and when it exits, the program prints out the total number of calculations performed. Ten students were brought in on a one-on-one basis, each having 60 minutes to complete this task. They were provided with an executable version of the program that they could run as many times as they wanted during the study. There were two rules for them as they attempted to reverse engineer (create the code to make the program work based on the executable) the functionality of the executable: 1) The three operations (addition, subtraction, and flipping the signs) must be implemented in separate functions. 2) The statements that print out the results of

the operations must be in the main function. During the 60 minutes, the students were tasked to think aloud while attempting to implement a solution to creating the calculator program. Both the screens and the audio of the students were recorded and analyzed as described in Chapter 5. After initial analysis using open coding techniques, we determined that we could analyze the data with a coding scheme based on well-defined concepts from the literature. The final categories for which we recorded data were: **Time**, **Utterance**, **Action Code**, **Current Task**, **Problem Solving Phase**, **CS Concept**, and **Proposed Certainty Level**. These are described in more detail below:

Time: This column contained the elapsed time from the start of the session to the start of the associated utterance or action. This allowed us to develop a timeline for when participants did certain activities such as saving, compiling code, running the provided executable, etc. Reviewing the videos for analysis, the time when a certain action started would be marked into this column on a spreadsheet. Times were not recorded for every utterance, just when actions started.

Utterance: This column contained the transcribed audio broken down to text representing either a full thought or partial thought.

Action Code: The potential actions were Compile, Save, Edit, Execute Own, Check Executable, View, Review, Tools, or Comment. Check Executable indicates that the participant executed the sample calculator program. Execute Own indicates that the participant executed their own compiled code. Tools refers to when participants discuss one of the tools used for coding such as the editor or a tool they have used in the past.

Current Task: The potential current tasks were Main (Menu), Main (Loop), Main (Counting Operations), Addition function, Subtraction function, or Flipping Signs function. The Current Task column was coded for whenever there was an utterance that focused on a specific CS concept or an Edit action when a participant started coding after performing some other action.

Problem Solving Phase: Based on McCracken's[132] framework for the learning objectives of novice programmers, we used the phases Understanding the Problem, Breaking Into Subproblems, Implementing Solution (Subproblem), and Implementing Solution (Combine). We separated the implementing solution phase to differentiate between a focus on solving the problem versus a focus on solving one the tasks identified in the Current Task codes.

CS Concept: The codes we used for this work were based on Goldman[78] and included Parameters/Arguments I/II/III (PA1/2/3), Procedures/Functions/Methods (PROC), Control Flow (CF), Types (TYP), Boolean Logic (BL), Syntax vs. Semantics, Operator Precedence, Assignment State-

ments (SVS), Scope (SCO), Abstraction / Pattern Recognition and Use (APR), Iterations/Loops 0/2 (IT0/2), Arrays I/II/III (AR1/2/3), Memory Model, References, or Pointers (MMR), Design and Problem Solving I/II (DPS1/2), Debugging / Exception Handling (DEH), and Other (OTH). This is not an exhaustive list of the concepts from that paper, but includes those that were captured in the assigned task. The Iterations/Loops 0 category was created by the authors using the same description as Iterations/Loop I, but replacing "nested loops" with "non-nested loops."

Proposed Certainty Level: This column was used to categorize our view of participants' confidence in what they did/said. The codes for this column were No Knowledge, Uncertain, Muddled, Certain (Correct), and Certain (Incorrect). These were used to categorize our view of participants' confidence in what they did/said.

Below are a summary of some of the key findings of the study after coding the 10 participants using the developed schema.

3.2.1 Pointer Issues

In the coding in the wild study, students exhibited issues related to the use and syntax associated with pointers in the context of parameter passing while trying to implement the flip signs function. For example, one of the participants in the study was able to eventually succeed at implementing the flip signs function, but used a trial-and-error approach while trying to implement pointers, going back and forth in his thinking trying to differentiate between the "&" and "*." This finding of issues with pointers is not novel. Lahtinen et al. and others found that pointers are a common struggle for introductory programming students[116, 23, 33]. This blind application of operators when dealing with pointers was also observed by Craig[43]. A likely reason for the existence of this misconception is again fragile knowledge. Students have learned pointers by the time they take CPSC 1020 (CS2), but have difficulty with the precise syntax.

3.2.2 Arrays

In the coding in the wild study, students exhibited difficulties with the use of arrays. In particular, using arrays as a means of passing by reference, or changing the value of a variable while in the scope of another function. We saw one student who, while implementing the flip signs function, attempted to declare an array with a return type of integer and then return an array

back to the main function. The eventual resolution of the compiler messages that he did not fully grasp was to avoid the complexity and find a workaround to implement the flip signs function. Such programming difficulties with arrays were previously seen by various computing education researchers[78, 100, 123, 33].

As with pointer difficulties, fragile knowledge is the theoretical basis for this difficulty. This can also be traced back to Fleury's work[65] as we observed that students seem to be constructing their own rules about how parameters are passed using arrays,and these rules were not accurate and caused errors when trying to implement the solution.

A suggestion to address Fluery's results of students constructing their own rules would be to have students engage in activities such as think-pair-share[63] and self-explain[30] to prompt students to directly address how they are conceptualizing parameter passing. In this way, incorrect mental models can be exposed and addressed earlier in the learning process.

3.2.3 Return Values

Students struggled with understanding how to return values from one function and store them in another function. For example, in the coding in the wild study, one student was never able to resolve the issue of capturing return values from the separate functions back in the main function, so he never successfully completed the task. This finding is not novel; return values are one of the important and difficult topics found by Goldman in his work[78]. Fragile knowledge is a likely explanation for the prevalence of this misconception.

3.2.4 Correct Output Without Full Understanding

The coding in the wild study revealed that some students were able to produce correct output through a combination of interacting with compiler feedback, trial-and-error, and complexity avoidance, but did so without fully understanding the concepts they were implementing or why the output worked the way it did. This was evident with students working on the flip signs function and seen whether it was implemented with pointers or arrays. For example, one student, while attempting to implement pass by reference semantics, engaged in trial-and-error and he explicitly said he was certain would be incorrect. He managed to achieve the correct output, even with expressed uncertainty that his solution was not correct.

This finding appears to be novel in the misconceptions literature. Although as described later, work has been performed that looks into trial-and-error approaches, this result of students not understanding the concepts that they are attempting to implement and still not having that understanding after successfully implementing them is novel. Quotes from expert reviewers for the Innovation and Technology in Computer Science Education (ITiCSE) conference at which this work was accepted provide evidence for the usefulness of this result: "The insights around solutions that "work" but are actually only working by accident and still based on incorrect understanding is particularly useful" and "the analysis takes into account distinction between mastering a programming concept and creating a correct code which is very valuable."

With respect to the approaches of students using trial-and-error to obtain the correct output, Buck and Stucki[21, 22] suggest that this strategy stays with CS students because of how the curriculum is implemented in most classes. Students generally are taught to focus on application and synthesis skills (writing code) on Bloom's taxonomy of learning[1]. Edwards claims that based on Bloom's taxonomy, comprehension and analysis must be mastered before students can effectively write programs using the application and synthesis skills[58]. He claims that students are not properly taught the skills to comprehend and analyze programs, which leads them to practice more of writing/implementing the programs, where they employ techniques such as trial-and-error and eventually manage to achieve the correct output but do not fully understand the concepts they work on. Fragile knowledge also plays a role in this observation. Students have learned certain concepts such as pass by reference semantics, but do not have a clear enough grasp on them to implement them successfully without compiler feedback and have to continually guess and check different routes.

CS instructors may benefit from an emphasis on understanding student processes and what they're thinking as opposed to a singular focus on the production of code that passes certain test cases. Krausel and other researchers have attempted to strengthen students' comprehension and analysis skills through code reading assignments and having students reason about and manipulate non-code artifacts[115]. There is also the "inside out" way of teaching CS1 students to program, which focuses on solving smaller problems first as opposed to immediately having students attempt to work at skills higher up on Bloom's taxonomy[21, 22]. These implications are particularly strong when considering how grading/feedback is managed in the courses. Edwards suggests and looked at using test-driven development as a way to offer immediate feedback and allow students to address the analysis and comprehension of programs early in the learning process[58, 57].

3.2.5 Planning/Problem Solving

Students in the coding in the wild study exhibited issues that resulted from how they planned or did not plan to problem solve. In particular, the problematic approaches involved attempting to implement the program in the order the executable was presented and tested, without stepping back to plan. This led to some students running into errors such as not including necessary function prototypes before the main method. The same was true of running into problems; students did not often think to stop and decompose the problem before just attempting to fix or address it after seeing a compiler error.

Similar to the above discussion on correct output without full understanding, this would seem to be curricular-based, with introductory CS courses focusing on having students write programs and not engage in the lower levels (comprehension and analysis) of Bloom's Taxonomy[1].

This result suggests that students should be given more opportunities to work on the planning phases that are useful and effective for programming. An implication for instructors would be related to the "inside out" method of teaching Buck and Stucki proposed[21, 22], which combined with active learning activities could allow students to analyze programs and work to figure out how a plan would be written for a given program as opposed to simply having prompts where students have to program without explicitly considering a plan. Edward's work with test-driven development would be another suggestion, as the development would have to be planned on the front-end to ensure that the implementation is compliant with the tests[58, 57]. Thinking about planning and determining a problem before attempting to solve it can allow for a deeper understanding of concepts and fewer misconceptions. Simple interventions such as prompting students to have to explicitly provide plans for how they would solve a programming problem either as standalone activities or a precursor before they have to attempt to implement solutions would be pedagogically beneficial to address this concern.

3.3 Software Engineering Design Documentation Interviews (SEDDI)

The third study was conducted in the context of Clemson's Software Engineering course. The course was project-based, with students divided into groups of three, typically self-chosen.

The project had two major assignments, broken down into a series of submissions. The objective of **Assignment one (A1)** was to develop the requirements for an Autonomous Vehicle Management System that supports both fleet administrators and end users in interacting with a fleet of autonomous vehicles. The system should be designed to make recommendations about fleet composition and manage deployment, booking and payment functions. The system should accommodate the needs of users who are visually impaired or hearing impaired. In A1, students had to go through a requirements analysis process that included preparing 10-15 appropriate questions to ask various stakeholders, interview the stakeholders and summarize answers to key questions, and provide multiple use case input/output scenarios to ensure they understood the functionality of the system. All materials were then submitted as a single professional requirements document for a grade. Students received detailed written feedback after each submission. For this study, submission A1 was not explicitly used for analysis, but was necessary to complete the rest of the project and came up as a source of difficulty for the students in the interviews.

Our evaluation of the approach of the student teams to design, the obstacles they encountered and the misconceptions they exhibited is based on three sources of data:

Assignment 2, part 1 (A2.1), the purpose of which was to produce a design of the system whose requirements they analyzed in A1. The goal of A2.1 was to identify a set of application-level classes that would be needed to develop the system and to show the relationships among the classes using one or more class diagrams. They were also asked to provide short, high-level descriptions of the functionality of each class in their designs. A1 had also asked them to describe the system inputs and outputs for important use cases. This typically meant an input of a rider requesting a ride through the mobile app interface and an output of a car arriving to pick them up, deliver them to their desired address and a payment occurring. Teams were also asked to provide preliminary visuals of what a rider and fleet administrator might see, and provide a summary of how their system design supports the interactions in the use cases and their visuals (screen mockups). Finally, they were asked to provide five other Unified Modeling Language (UML) diagrams (some combination of sequence, state or activity diagrams) to clarify aspects of the design corresponding to key risk areas or features that seem hard to get right. Students submitted these documents for detailed feedback but did not receive a grade. However, A2.1 is useful to examine in that misconceptions and design difficulties may still manifest.

Assignment 2, part 2 (A2.2) asked the teams to refine and improve all their answers to

part 1: to elaborate on the functionality of the classes in the class diagrams by describing public interfaces, including attributes and operations, to revise and improve their visuals, descriptions of inputs and outputs, and design verification (summary of how the system supports the use cases seen in the visuals and represented in the provided example inputs and outputs). They were asked also to provide descriptions of key operations in the form of informal requirements and guarantees. A2.2 was used as a final comparison to A2.1 to observe the changes that were made based on feedback provided.

In addition to these documents, the groups were asked to participate in interviews that I led to discuss their design decisions. The students were asked to do the following:

1. Explain their UML class diagrams.
2. Walk through a standard use case of their system.
3. Walk through a non-standard use case of their system (e.g. A visually or hearing-impaired person using the system or a car needing maintenance).
4. Describe some of the challenges faced in the project throughout the semester.

Groups had their project documentation on the screens and could use it to help explain their answers. The screens as well as the audio from the interviews were recorded. Preliminary analysis was performed and the full analysis will be completed using open coding strategies to identify emergent themes in the interviews. Key results based on preliminary analysis of the answers to the questions and comparing them to the submitted documentation is shown below:

3.3.1 Incompleteness and Inconsistencies

In examining the SEDDI interview transcripts and comparing those to the design documents, we found groups that had designs that did not fully capture the required system functionalities (incomplete) or elements/functionalities that were described in user scenarios or interview transcripts but were not supported by the design diagrams (inconsistent). An example of an inconsistency is found in one group who mentioned in their interview that a "beep would come from the phone" like a "metal detector" as a way to alert visually impaired users that their car had arrived. This feature was not present in the design documentation in any way. An example of an inconsistent design is a

group that had user interface designs showing a history of transactions but this was not present in the class diagram or any other diagrams.

3.3.1.1 Theoretical explanations

This result seems to stem from students not properly communicating as a group. There were groups who would divvy up the parts of an assignment and it was apparent that they did not communicate to ensure that the parts were consistent or completely finished across the document. The theory here relates back to social constructivism[209], and giving students a chance to create their own knowledge not just individually, but interacting together as a team. In regards to the inconsistencies exhibited during interviews, my theory goes to the work of Kahneman discussing system 1 and system 2[101]. Students seem to have been intuitively replying to questions with features they believed they had designed or they felt made sense.

To address this issue, instructors should emphasize the importance of being consistent in planning and design. Allowing students to go through good and bad examples of system design consistency and completeness and discuss and critique those designs would help ensure students did not make the same mistakes. Also, encouraging the teams to communicate and review the design decisions together, potentially requiring them to explicitly write up feedback for design decisions would promote social construcvism of this knowledge.

3.3.2 Modeling Abstract Concepts

Many groups struggled with how to represent elements of the project that were not concrete or did not have an obvious physical representation in the real world.

3.3.2.1 Evidence

The most common example of this was shown in attempting to create a "Ride" class. The class diagrams generally had correct examples of a "Car" class, which is a physical object, but either would have no representation of a ride in their system design or the representation would be incomplete or inconsistent with what the groups explained in their interviews and wrote in their use case scenarios.

Student difficulty with abstraction is not novel, and has been reported by Thomasson[204] as a common OO misconception and was also discussed by Or-Bach and Lavy[144]. However, it was

interesting to observe this phenomena at the design stage before any programming occurs.

Detienne claims that an issue with abstraction is one of decomposition[50]. Novices have trouble breaking down a larger unit into smaller, functional units which makes it hard to design abstract concepts. Or-Bach and Lavy found that many students struggled with abstraction, and claimed that the reason stems from the focus being the structure of object-oriented design as opposed to the process[144].

When teaching elements of design or implementation, instructors may wish to to provide multiple examples of functions, classes, and other concepts that represent abstract concepts. Or-Bach and Lavy suggest "discussions in which students communicate, present and evaluate different approaches to solving complex problems can develop their sense of criticism towards quality of solutions."[144]. They agree with Machanick in showing students existing abstractions to use as building blocks for their knowledge of the concept before having them design their own abstractions[128]. Pennington et al. suggest explicitly showing the differences between novice and expert behaviors when it comes to object oriented design[155]. There is a general logic that makes sense for this result where it is easier to represent things that can be touched and broken down into components or elements. Beginning with physical items would be a good way to introduce students to the idea of representation, but the abstract elements need to also be explicitly discussed and practiced, through ways that allow students to properly construct knowledge (active learning techniques, employing social constructivism, etc.).

3.3.3 Importance of Feedback/Reflection

A key finding of the formative studies was that students valued feedback and the opportunity to reflect on their work. Feedback for the EYR + TBIs studies came in the form of me being able to ask students to walk through code line by line or explain something further during the TBIs. For the coding in the wild study, feedback was mainly from syntax feedback by observing compiler errors and warnings. Students also had feedback from themselves, as the think aloud protocol allowed them to perform some self-explaining while working through their implementations that generally would not occur if a student were coding in silence. For the software engineering interviews, feedback was explicitly mentioned as being provided by the instructor between various submission points of the project. This helped them better understand the concepts they were working on.

The EYR study showed that some students who answered a question incorrectly then answer

a similar question correctly with correct reasoning later on in the same assessment. This shows improvement after getting to examine a problem, write out and reflect on an answer, and then address a similar problem soon after. The students being able to construct their own knowledge allows them to learn and recognize an error that they previously did not.

Also, the TBI participants were generally able to realize misconceptions and correctly explain code snippets when asked to explain the code line by line. This is explainable by the social constructivist framework, with language and having the ability to interact with a person (myself in this case) through the use of feedback helping students to see and address mistakes in their thinking.

The coding in the wild study saw students using compiler feedback as a tool to correct and fix errors. These were sometimes oversights, other times misconceptions that students were able to address and discuss where they went wrong while being recorded for the think aloud protocol. This interaction with the compiler combined with the think aloud protocol allows the students to construct their own knowledge and talk through misconceptions.

Students in the software engineering interviews explicitly stated the value of feedback in the project and its various phases. The value of reflection was seen as students looked at their design documentation and used it while walking through use case scenarios or describing their UML class diagrams.

The importance of feedback has been documented in the research. Gusukuma looked into the idea of having a compiler that gives feedback based on common misconceptions[80]. Brown[17] and Roediger[170] found additional benefits for active learning when students received feedback. Autograders are also growing in use to allow more feedback to be provided as enrollments increase. As mentioned in Chapter 2, Higgins found that students value feedback and feel as if they are owed it as consumers of knowledge[87]. These effects can be explained by the theories of constructivism, social constructivism, and documented benefits of active learning.

Provided in the right area, time, and guided towards the proper source of difficulties/misconceptions, feedback and reflection can allow students to address their struggles early in the learning process. Incorporating elements of constructvism and active learning techniques such as Chi's self-explaining[30] should also be beneficial. Feedback is the result that is focused on for my final evaluative study. This element, combined with an appropriately designed active learning technique, potentially offered benefits for learning outcomes and addressing misconceptions as it was a persistent result through the three formative studies.

3.3.4 Complexity Avoidance

An observed result throughout my formative studies was students encountering difficult concepts or tasks and instead of addressing them, choosing to avoid them, sometimes in creative ways through finding workarounds, other times through simply choosing to ignore the issue. In my code in the wild study, students would start trying to apply pointer logic to the flip sign function but then find workarounds to produce outputs that were correct but avoided pointers, or in some case, mentioned how they could use pointers but preferred to avoid them and never even tried to implement a solution with them. There was an instance of a student having the flip sign function return a value and just calling it twice in the main function to flip both integers. There were also times when students started attempting pointers but then decided to use arrays instead to solve the problem.

Comparing the software engineering interviews with the design documentations, I observed many groups attempting to avoid dealing with functionality that they considered too complex. Sometimes, they would make use of a megaclass or database that had unexplained functionality and use that as the explanation of the complex concepts that they had not figured out how to show.

The students use of a *megaclass* to avoid complexity relates to work on software design antipatterns such as the "God class"[168] and the "blob"[18]. Smith claims that these antipatterns can stem from attempting to design procedurally but masquerading it as object-oriented[184]. Lawson also observed students avoid complexity while attempting to solve a concurrency problem[119].

Brown's book on antipatterns[18] offers multiple explanations for a strategy such as the "blob" or "God class." Among them are not having an adequate understanding of object oriented principles or appropriate abstraction skills (another observed difficulty), which would equate to fragile knowledge for the software engineering class. Another explanation by Brown is the architecture design not being enforced, sometimes stemming from not reviewing the architecture properly, which he claims is "especially prevalent with development teams new to object orientation." This explanation would be very applicable to our software engineering course and would tie back to social constructivism, or lack of social constructivism, as many groups struggled with communication, which hindered them from ensuring their designs were adequately reviewed.

This phenomena also seems explainable by Kahneman's S1 and S2[101]. When designing the diagrams, students might have been using S1 and unintentionally avoiding complexity by over-

simplifying the problem in their minds. An alternative theory is that students may have been using the deep thinking S2 and used strategies such as complexity avoidance to work around the lack of an apparent and intuitive simple solution.

This finding supports the idea that students need to have difficult concepts reinforced and additional focus on the learning of CS concepts that students find complex or do not have a firm grasp on. Active learning techniques such as Chi's self-explain[30] to ensure that students understand concepts that are complex and ensuring that students have the opportunity to receive feedback on their conceptions about complex issues would be useful. Also, encouraging the teams to communicate and review the design decisions together, potentially requiring them to explicitly write up feedback at certain phases of the project would aid in addressing the issue of inadequate reviewing.

3.3.5 Trial-and-error Design Approach

In the coding in the wild study, students would often get to a place where a functionality was not working in the intended way and they were not quite sure of the solution. The students would have a few ideas and begin using trial-and-error until they were able to produce the correct output. When attempting to implement the flip signs function using pointers, students would receive compiler feedback about reference types and as the feedback was not fully understood, they would go into their code and arbitrarily swap out "*" and "&" at various places they believed would fix the problem. Software engineering design documentation compared with the interviews showed instances of groups attempting to design their class diagrams through trial-and-error of either feedback they would receive from their instructors or knowledge they would gain from class. This phenomenon has been reported previously. Hundhausen[95] has reported on students using workarounds and trial-and-error while programming.

As mentioned earlier, Buck and Stucki[21, 22] suggest that this strategy stays with CS students because of how the curriculum is implemented in most classes. Students generally are taught to focus on application and synthesis skills (writing code) on Bloom's taxonomy of learning[1]. Edwards claims that based on Bloom's taxonomy, comprehension and analysis must be mastered before students can effectively write programs using the application and synthesis skills[58]. This result also stems from students having fragile knowledge of the concepts they are trying to implement.

Edwards suggests moving away from trial-and-error approaches for teaching CS and considering a test-driven development approach[58]. This does not seem completely ideal as a lot of

the actual programming process does occur from trial-and-error and it is not inherently bad as a technique. As mentioned earlier, I believe addressing the conceptual issues and improving the lower level Bloom's taxonomy skills through means such as code reading assignments and having students reason about and manipulate non-code artifacts as suggested by Krausel[115]. There is also the "inside out" way of teaching CS1 students to program, which focuses on solving smaller problems first as opposed to immediately having students attempt to work at skills higher up on Bloom's taxonomy[21, 22]. Instructors should be aware of the cons of the the trial-and-error approach and show explicit examples of how this can lead to cases where the code output is correct but there are errors in the implementation of the code or conceptual errors, allowing students the opportunity to discuss these examples and errors with each other.

Chapter 4

"Explain Your Reasoning" Survey + Task-Based Interviews

This paper was published at the 2018 Koli Calling International Conference on Computing Education Research held in Koli, Finland under the title "What Are They Thinking?: Eliciting Student Reasoning About Troublesome Concepts in Introductory Computer Science." [104]

4.1 Introduction

Computer science pedagogical content knowledge (PCK) refers to the "blending of content and pedagogy into an understanding of how particular topics, problems, or issues are organized, represented, and adapted to the diverse interests and abilities of learners, and presented for instruction[177]." Understanding student conceptions and identifying student misconceptions is an important element of PCK and a precursor to the development of high quality pedagogical materials. Such insights into student reasoning are also of use in fleshing out the related notions of fundamental ideas[176], threshold concepts[134, 173, 55, 64], and liminal spaces[203].

We ask several questions: What topics do students in an introductory C-based computer science course find troublesome? What misconceptions do students have and why? How do these topics relate to prior work in misconceptions in CS education? What interventions might be used to aid in addressing these troublesome concepts? It is worth noting that identifying the full set of

troublesome concepts will require a series of studies. We report here on an initial study of student reasoning and misconceptions about selected topics.

In selecting topics for study, we build on prior work addressing the identification of threshold concepts[173, 55], fundamental ideas[176], "important and difficult"(I&D) topics[78], and troublesome concepts[156]. In constructing our survey we build on related work in concept inventories[23, 201, 148, 86], other assessments[132, 123] and guidelines for assessment[127]. In analyzing our results and placing them in context, we rely on related work that explores student reasoning about these topics[208, 169, 156] and student misconceptions[91, 43, 181].

We developed and administered a survey consisting of a demographic questionnaire and 21 content-based questions to 106 students at the start of a second semester course for CS majors (i.e., CS2) at a large public U.S. university that focuses on engineering and science. Our survey uses questions designed to probe student understanding of key concepts. We include closed-form questions as in concept inventories (CI)s, but also draw from the formative stages of CI creation and the intent of think-aloud interviews, probing further by asking students to provide explanations of their reasoning about the questions. Student responses are evaluated as either correct or incorrect. We then select the questions that appear to be most problematic for students and several closely related questions and engage in open coding of the students' explanatory text to expose the details of student thinking and detect emergent themes.

By correlating student responses and explanations across multiple related questions, we gain insight into whether students consistently hold and apply their expressed conceptions or whether they are applying strategies related to information foraging[161, 118] or abduction[122], in which their interpretations of the behavior of the program is based on contextual clues such as the names of variables and functions, and the apparent purpose of the code snippet.

Some researchers have followed up with think-aloud interviews to capture student misconceptions[100, 136, 14, 75, 62]. We follow up with a related technique, task-based interviews (TBIs), to clarify and correct the understanding of student conceptions formed through analysis of the surveys. Results of the interviews point to several factors as causes of the apparent misconceptions, among them fragile knowledge about parameter passing, lack of exposure to the use of global variables, and abductive reasoning in the presence of misleading contextual clues.

In the following section we discuss the background that inspired this research. In Section 4.3 we describe the design and implementation of our study and in Section 4.4 we provide and

attempt to explain the qualitative and quantitative results. Finally, we conclude with a discussion of contributions, limitations and future work.

4.2 Background and Related Work

Pedagogical content knowledge (PCK) for computer science is important but currently underdeveloped[93]. Instructor knowledge of what students get wrong, their misconceptions, learning difficulties, and symptoms of misunderstanding is a critical piece of PCK[172]. PCK for computer science is based upon education research performed in the CS context and disseminated in the CS education community. Additionally, instructors may possess refinements of that PCK formed through their individual experiences.

Researchers have looked into notions related to PCK, such as threshold concepts. Troublesome knowledge, an important characteristic of threshold concepts, is knowledge that appears "counter-intuitive, alien, or incoherent" in light of prior knowledge[134]. Troublesome knowledge can take multiple forms (ritual knowledge, inert knowledge, conceptually difficult knowledge, alien knowledge, tacit knowledge, and troublesome language) and derive from multiple sources [156]. This related work in fundamental ideas in computer science [176], threshold concepts [134], and "important and difficult" topics[78] are a good starting point for selecting concepts to explore. Related work in troublesome knowledge may provide a basis for understanding why students possess certain misconceptions[156].

A concept inventory (CI) is a criterion-referenced test, designed to help determine a student's knowledge of a specific set of concepts, and to expose misconceptions. Taylor et al. describe the development of CIs for computer science as being "in its infancy"[198]. Tew and Guzdial developed the FCS1[201], a validated instrument for CS1, that uses pseudocode in an effort to be applicable across a variety of pedagogies and programming languages. Parker et al. replicated this in an "isomorphic" form as the SCS1 to enable broader distribution and avoid saturation[148]. Decker designed an assessment for CS1 and CS2 in Java[48], which was developed and tested at her home institution as part of her thesis work. Caceffo describes his process for the development of a concept inventory for introductory programming using the C programming language [23]. The creation of concept inventories relies on the selection of appropriate topics for inclusion.

Work performed by Goldman et. al focused on the selection of topics that the authors term

"important and difficult" (I&D)[78]. The authors use a process that involves having a group of experts (people with years of experience teaching introductory CS courses and who have written pedagogical articles or textbooks on these subjects) propose important topics in introductory CS. They then rate these concepts on a 1-10 scale in terms of importance and difficulty. Next they negotiate and re-rate the concepts, providing an explanation if they choose a rating outside the initial ratings' inner-quartile ranges. These justifications are then anonymously shown to the experts and they are asked to do a final rating. This process produced 11 topics that the authors deemed I&D. Other CS education researchers have examined what topics are difficult or error-prone for students, with [43] looking at difficulties with pointers in the C language, [28] using a large dataset of CS1 students solving programming problems to identify challenging concepts, and [91] providing a detailed list of common syntax and logic errors observed in an introductory Java programming class.

STEM Researchers have looked into misconceptions and how they can affect learning in various subjects [112, 42, 81, 84, 89, 26]. Confrey [38] states that even before formal study, people have "firmly held, descriptive, and explanatory systems" that are different than what's in the curriculum and "are resistant to change through traditional instruction"[111, 147]. Tew and Guzdial posit that this may be less true for computer science, and that misconceptions in this domain may be more related to aspects of instruction rather than due to beliefs that students bring with them[201]. Confrey found that within programming, misconceptions stem from systematic errors. Researchers have explored misconceptions on topics including propositional logic[86], memory models and assignment upon declaration[100], algorithms and data structures [152], looping strategies [187], BASIC programming statements [8], language-independent conceptual "bugs" in novice programming [153], and misconceptions and attitudes that interfere with learning to program [33].

Other assessments exist that attempt to uncover student conceptions but are not technically "concept inventories." Drachova et. al developed a comprehensive inventory of principles for reasoning about correctness of software[54], using the work of [100, 78] and others to develop a framework to support teachers by providing a structured set of learning outcomes and a way to assess if students have achieved these outcomes. McCracken et. al broke down the learning objectives for first year CS students into 1) Abstract the problem from its description; 2) Generate sub-problems; 3) Transform sub-problems into sub-solutions; 4) Re-compose the sub-solutions into a working problem; and 5) Evaluate and iterate. These researchers found that the performance of first year students was

significantly worse than expected [132].

In comprehending a new program, it is documented that novices and experts behave differently [169]. Novices can struggle with things such as how complex algorithms are in certain languages and how fragile their knowledge can be at first. Some of the issues with novices relate back to the troublesome language discussed by Meyer [134] and Perkins[156]. Robins et al.[169] suggest that motivation and getting students to gain knowledge related to programming first would be beneficial to make "more effective novices." They also suggest explicitly focusing on programming strategies in introductory programming courses, rather than focusing on the syntax and semantics of programming language features and teaching students how to properly design basic programs to give them the ability to tackle problems regardless of the specific features of the language used. These suggestions are consistent with the observations of Soloway that the major stumbling block for novices learning to program is in "putting the pieces together, composing and coordinating components of a program" rather than in language constructs[186].

In conducting this study with students in their first year of study of computer science, we needed to select challenging and important topics, design and pilot a survey instrument, administer the survey, and then analyze the results. The methodology employed in each of these processes is described in the subsections below.

4.3 Experimental Design

4.3.1 Selecting the topics

To select topics for the study described in this paper, we began by asking what concepts are included in introductory Computer Science (CS) courses. To answer this question, we first examined resources that identify concepts generally covered in CS courses: the ACM curriculum guidelines[99], and related work on concept inventories for computer science[148, 201, 198]. We then looked for topics at or near the threshold for classification as "important and difficult" by Goldman, et al.[78] and intersected all of these with considerations of local context: the topics taught in the first semester computer science course for majors at the university at which the study was conducted, and the outcomes of instructor interviews at that institution.

We examined the syllabus used in the institution's CS1 course, which uses the C programming language. This syllabus broke down the topics into two modules, each roughly corresponding to

half a semester. Module one, which this paper focuses on, addresses the topics of: number systems, variables, arithmetic operations, loops, user input, conditionals & Boolean variables, functions and arrays.

We obtained old CS1 exams from professors who recently taught or currently teach the course and identified concepts covered in the exams. We interviewed the faculty members, asking "Reflecting on your most recent offerings of CS1, can you identify the topics that students found most difficult? If so, what are they?" and " Why do you think that topic was difficult for students?" As we developed questions for inclusion in the survey, instructors were also asked if students would find the particular question difficult and why.

Based on this review of the literature and comparison to the course syllabus, we selected five main I&D topics, listed below. We include in the listing the corresponding codes from [78] and their ratings for importance (I) and difficulty (D). These ratings (shown below) are from 1-10 and were determined using the Delphi process described in Section 4.2.

1. Parameters/Arguments:
   (a) Call by Reference v. Call by Value: Understanding the difference between "Call by Reference" and "Call by Value" semantics (PA1: I = 7.0; D = 7.4)
   (b) Formal v. Actual Parameters: Understanding the difference between "Formal Parameters" and "Actual Parameters" (PA2: I = 8.6; D = 5.7)
   (c) **Parameter scope, use in design**: Understanding the scope of parameters, correctly using parameters in procedure design (**PA3**: I = 9.1; D = 7.5)
2. **Procedures/Functions/Methods**: (e.g., designing and declaring procedures, choosing parameters and return values, properly invoking procedures) (**PROC**: I = 9.8; D = 9.1)
3. **Scope**: (e.g., understanding the difference between local and global variables and knowing when to choose which type, knowing declaration must occur before usage, masking, implicit targets (**SCO**: I = 9.4; D = 8.0)
4. Assignment Statements: (e.g., interpreting the assignment operator not as the comparison operator, assigning values from the right hand side of the operator to the left hand side of the operator, understanding the difference between assignment and a mathematical statement of equality) (AS: I = 9.5; D = 4.4)

5. Control Flow: Correctly tracing code through a given model of execution (CF: I = 9.8; D = 7.0)

These five topics are all covered in the CS1 course under study. The first three (PA1/2/3, PROC, and SCO) are all covered within the "functions" section of module one. Those labeled in bold (**PA3, PROC,** and **SCO**) met the threshold to be considered important and difficult by Goldman and the experts who participated in the Delphi process. The final two, Assignment Statements and Control Flow were both rated highly important in the [78] study, but were not rated significantly high in terms of difficulty. We added these based on instructor reports and interviews and related work. Specifically, work by [123] indicates that students have difficulty with control flow and that it is important and work by [39] suggests the same is true of assignment statements. Instructor reports indicated that some students still struggle with these topics in introductory courses, particularly with interactions between assignment and control flow.

4.3.2 Designing the Survey Instrument

To study these concepts of interest, we developed a survey intended to gauge how well introductory CS students understand them. We drew on work from concept inventories for fundamental CS concepts[201, 148, 198], as well as other attempts at standardized tests of CS concepts[100, 132]. Our final survey has questions that contain code snippets and ask students to freely respond with the output or to select between two choices (one correct and one distracter). The distracter answer (based on misconceptions suggested by the literature or interviews), was determined by using code snippets that did not offer many alternatives for a correct numerical answer. For example, a question with a subtract function had arguments passed in a certain order (subtract(y, x)). The correct answer choice would be the value of y - x, whereas the distracter would be the value of x - y. For both types of questions (multiple choice and free response) students are then asked to explain the reasoning behind their answer. The goal of collecting this qualitative data is to help us understand students' thought processes and to expose misconceptions.

Our original survey consisted of a vocabulary matching assessment and a set of 33 questions in the format described above. We ran a pilot of this survey in a 3rd year undergraduate CS course and observed that most students could not finish in the allotted time. We decided to eliminate the vocabulary assessment completely and to reduce the number of questions in the survey.

The information obtained from the previous exams and faculty interviews allowed us to shorten the survey down to its final version of 21 questions. Elimination of questions was based on a number of factors, including removing questions that focused more on syntax as opposed to an actual conceptual problem, removing questions that did not seem to fall in line with previous exams, and removing questions that professors did not feel would be difficult or important. Listed below are the concepts that are the focus of each of the 21 survey questions. Those listed in bold are discussed further in the paper.

1. Modulus Operator
2. Difference Between Pre and Post-Incrementer
3. Array Manipulation (Array index can be an expression)
4. Assignment is not the same as checking for equivalence
5. Order of evaluation of operators
6. Multiple conditions in if statement
7. Properly tracing through code linearly
8. Properly tracing through code with loops
9. Understanding parts of function prototype (Return type)
10. Understanding parts of function prototype (Parameter type)
11. **Scope of variables in function (function only affects local variable); Pass-by-value**
12. Order of parameters/arguments in functions matters
13. 1to1 mapping of parameters passed to parameters used in function
14. **Function with a void type doesn't return a value; scope of variables**
15. Functions can be invoked with primitive values
16. Latest variable assignment
17. **Function with a void type doesn't return a value; scope of variables**

18. **Global variables are visible whenever there isn't a local variable of the same name**
19. Array parameters in functions (Arrays are passed by arrayName)
20. If return value of function is not stored, it will be lost
21. A variable's scope, lifetime, and visibility within a program; Multiple declarations of a variable with the same name in different functions

Below we present and discuss the four questions of interest for this paper. For each question, we included line numbers so that students could easily reference specific lines when writing out their reasoning. Although the space below the "Explain your reasoning:" instruction appears limited in the diagram, students had half a page of blank space to provide a response during the study.

Question 11 includes a void function **swap** that swaps two values. The **swap** function exchanges the parameters x and y via a local *temp* variable. The function contains a return statement but is void and no value is returned. The **main** function initializes two variables, *cat* and *dog* to the values 5 and 8, respectively, and then invokes the **swap** function, passing in the values of *cat* and *dog* as parameters. However, since the parameters are passed by value, the value of *cat* never actually changes. Students are asked what the value of the variable *cat* will be after the function returns. They are given two choices: 5 (correct), in which the value of *cat* does not change and 8 (incorrect), which would result if pass by reference semantics were in effect rather than pass by value semantics.

Question 14 involves a void function, **findArea**, which defines a local variable *area* and then assigns to *area* the product of the *length* and *width* parameters. In the **main** function, a separate variable *area* is defined and initialized to 0 and variables x and y are defined and initialized to 4 and 8, respectively. **findArea** is then called, passing x and y as parameters. As in question 11, the function is void and no value is returned. In **main**, a print statement utilizes the *area* variable. Students are asked what result of the execution will be. The correct answer is that "The area of the shape is 0" should display.

Question 17, nearly identical to Question 14, provides a consistency check across questions. The function **subtract** accepts two integers as parameters, computes the difference and assigns the

Figure 4.1: EYR Question 11

Q11: Consider the following code segment:

```
01. void swap(int x, int y) {
02.     int temp;
03.     temp = x;
04.     x = y;
05.     y = temp;
06.     return;
07. }
08.
09. int main( ) {
10.     int cat, dog;
11.     cat=5;
12.     dog =8;
13.     swap(cat, dog);
14.     return 0;
15. }
```

What is the value of variable cat after the swap function returns?
a. 5
b. 8

Explain your reasoning below:

result to local variable *answer*. The **main** function also declares a variable *answer* and initializes it to the value 7. A print statement is called using the *answer* variable within the scope of **main**. The correct answer is b, since the *answer* variable that exists within **subtract** is a different variable than the *answer* variable in **main**.

Question 18 introduces a global variable, *answer*, initialized to the value 7 in line 1. The function **subtract** takes in two integers and returns an integer. Function **subtract** initializes a local variable *answer*, sets it equal to the difference between parameters x and y and returns the local *answer* to the **main** function, where it is assigned to the local variable *solution*.

Line 14 has a print statement that prints *answer*, with the intended *answer* being the global variable, as the *answer* variable from the **subtract** function is out of scope. The answer choices provided allow students to either choose the result of **subtract** (5-8), or the value of the global variable *answer*, which is 7.

Figure 4.2: EYR Question 14

Q14: Considering the following code snippet below:

```
01. void findArea(int length, int width) {
02.     int area;
03.     area = length * width;
04.     return;
05. }
06.
07. int main () {
08.     int x, y, area;
09.     x = 4;
10.     y = 8;
11.     area = 0;
12.     findArea(x, y);
13.     printf("The area of the shape is %d", area);
14.     return 0;
15. }
```

What is the result of the execution of this code?

Explain your reasoning below:

4.3.3 Administering the survey

The survey was administered to students who had completed CS1 or an equivalent introductory course and were currently enrolled in one of two CS2 sections, both taught by the same instructor. The instructor was not involved in this research project. The survey was administered by one of the authors during class time on a day the instructor was absent from the class. Students had the option of participating in the survey or an alternative assignment of equivalent effort worth one point of extra credit. The extra credit point was given regardless of how the students performed on the survey, as it was not graded. All students decided to take the survey, so an alternative assignment was never administered. The students ranged from freshman to senior, with a majority being freshmen or sophomores. Students ranged in age from 18 to 27; 83 were male, 21 were female, and 2 preferred not to report. Reported majors included BS/CS(55), BA/CS(15), BS/CIS(13) and Other (23; 19 of whom majored in a STEM field).

University IRB approval was obtained for the study and relevant consent processes were conducted. Students were given 50 minutes to complete both a short demographic questionnaire and the 21 questions. An example question at the beginning of the survey included a sample re-

Figure 4.3: EYR Question 17

Q17: Consider the following code segment below:

```
01. void subtract(int x, int y) {
02.     int answer;
03.     answer = x -y;
04.     return;
05. }
06.
07. int main( ) {
08.     int answer;
09.     answer = 7;
10.     subtract (8, 5);
11.     printf ("Answer is %d", answer);
12.     return 0;
13. }
```

What is the result of the execution of this code?
a. Answer is 3
b. Answer is 7

Explain your reasoning below:

sponse for the "Please explain your reasoning" portion of the question, meant to provide students a guide to the desired level of detail in their responses.

4.3.4 Task-Based Interviews

After administering and analyzing the survey, we developed a protocol for task-based interviews (TBIs). These interviews were conducted on a subset of students in a 2nd year undergraduate CS course. Students were compensated for their participation in the interviews. Before the interviews were conducted, students signed a consent form and were informed that the purpose was for them to think aloud while walking through various code snippets. Each interview was conducted one-on-one and video recorded for purposes of analysis. For each task, students were asked "What happens in this code snippet?" Depending on how students responded, the interviewer asked follow-up questions to probe at what students were really thinking. All four of the questions described in this paper were used for the TBIs, along with four other tasks. For this reason, no student who had participated in the survey was allowed to participate in the TBI process. The interviews were

Figure 4.4: EYR Question 18

Q18: Consider the following code segment below:

```
01. int answer = 7;
02.
03. int subtract(int x, int y) {
04.     int answer;
05.     answer = x -y;
06.     return answer;
07. }
08.
09. int main( ) {
10.     int cat, dog, solution;
11.     cat = 5;
12.     dog = 8;
13.     solution = subtract (cat, dog);
14.     printf ("Answer is %d", answer);
15.     return 0;
16. }
```

What is the result of the execution of this code?

a. Answer is -3
b. Answer is 7

Explain your reasoning below:

transcribed, and then the available text, video, and audio data was analyzed to gain insight into student reasoning.

4.3.5 Responses

Student responses and explanations were transcribed into electronic form. An initial analysis was performed to discover which problems students found most challenging. Questions 11 and 18 were determined to be problematic for students, with only 56 percent and 70 percent correct, respectively. As seen in the list in section 4.3.2, these questions addressed topics covered under "functions" in module one of the CS1 course: pass by value semantics and visibility of variables. We compared these two questions to the others in the survey and determined that Q14 and Q17 covered concepts related to those in Q11 and Q18: functions with void return types and scope of variables. In this paper we focus on the responses to these four questions.

4.3.5.1 Organizing the responses

We created a grid for each group of students with the same set of correct/incorrect answers. These grids included the unique identifier for each student, student explanations for their responses, and our notes about their explanations. We worked through the student explanations line-by-line and applied open coding, marking key phrases and concepts and identifying emergent themes. This work was performed iteratively, each author independently coding, then comparing notes, and then circling back to re-code as we uncovered new concepts, some of which were misconceptions that contributed to incorrect responses. Through the grid of responses with markings and notes we were able to cross-reference incorrect and correct responses with the goal of determining if students were systematically applying particular conceptions or misconceptions, if they were learning in the course of completing the survey, or if they were merely guessing. We also looked for "information foraging" behaviors[118] or evidence of *abductive reasoning*, which Letovsky describes in the context of program comprehension as "a plausible inference technique that involves explaining phenomena by using deductive rules backward to generate possible explanations"[122]. That is, we tried to determine if students were relying on their knowledge of the semantics of programming language constructs, or if they were using contextual clues such as the name of a method ("swap") to generate incorrect explanations about the semantics of parameter passing.

We organized the student responses into batches, based on the questions for which students had incorrect responses. Table 8.9 presents an overview of the results. Of the 106 students, 68 answered at least one of these four questions incorrectly. The "x" within a cell represents that question being answered incorrectly; the "Num Students" column gives the number of students who answered that particular grouping of questions incorrectly; the "Concept Codes" are abbreviations for the various misconceptions found. These codes are explained in more detail in the next section. The final row of the grid shows the total number of students who answered the individual questions incorrectly.

4.3.5.2 Concept Codes

After identifying misconceptions that students held for the questions of interest, we saw that they could be grouped into categories. We developed concept codes to more conveniently organize this information. The concept codes used in the table are defined as follows:

- PBV: Pass by value. Parameter passing using pass by value semantics passes a *copy* of each argument (actual parameter). The value of that copy is assigned to the local variable named in the function header (formal parameter). Functions with pass-by-value parameter passing cannot change the original value in the calling code. The PBV code indicates that students had misconceptions about the semantics of pass by value.

- GlX: Global Variable. A global variable is a variable declared outside of all functions. It is visible in all other functions and blocks unless that function or block has a local variable of the same name. The GlX code indicates that students had the misconception that global variables may not be accessed from the main function.

- OW: Overwrite (False-sharing). This is a specific type of misconception about scope in which students believed that a variable with the same name but in a different scope could be overwritten by a write to the local variable.

- GlOW: Global Overwrite (Variable shadowing). This is a more specific type of OW misconception, in which students had the belief that a global variable could be overwritten by modifying a local variable of the same name (i.e., in the presence of variable shadowing).

- N/A: Not Applicable. This means that there was no misconception that we could identify based on the reasoning students provided.

Table 4.1: Incorrect Answers on Questions of Interest

Row	Q11	Q14	Q17	Q18	Num Students	Concept Codes
1	x	x	x	x	5	PBV, GlX, OW
2	x	x	x		2	PBV, OW
3	x	x		x	9	PBV, OW, GlX, GlOW
4		x	x	x	1	PBV, OW
5	x	x			4	PBV
6	x			x	2	PBV, GlX
7		x		x	1	GlOW
8	x				25	PBV
9		x			2	N/A
10			x		2	N/A
11				x	15	Gl, GlOW
Tot	47	24	10	33	68	

4.4 Analysis

In the following section, we examine the results of the surveys and task-based interviews (TBIs) grouped by the observed misconceptions. For each misconception, we first explain the code, then provide student responses from the survey and our interpretation of those responses. Where data is available, we provide responses from TBIs for additional insight. We conclude each section with a subsection discussing explanations for these observed aspects of student reasoning.

4.4.1 Pass by value (PBV)

Again, parameter passing using pass by value semantics passes a *copy* of each argument (actual parameter) and the function is not able to modify the original value in the calling code. In Question 11 we observed student responses that indicated that students had misconceptions about the semantics of pass by value.

Student Response: "swap function simply swaps 2 values using a temp variable to hold one value in memory while it is replaced by the other" and **Interpretation:** Student believes that the use of a temporary variable in the swap function allows the effects of the swap function to propagate, even though the function is PBV.

Student Response: "swap will take the two inputs and reassign them so cat will become dog and dog will become cat" and **Interpretation:** Student believes reassignment of the inputs is happening in this PBV function.

Student Response: "I saw that it was a swap function at the top, so I assumed it did what its name was. After that, I solved the main, checking to see if y'all might have tried to trick us by reading the swap function. Then I looked at cat, and picked the number it wasn't equal to" and **Interpretation:** This student seemed to originally trust the name of the function and assumed it did what the name said. The student then checked to see if there was a trick in the main function, but managed to miss that the swap function was PBV.

A student had a similar response in the context of a task-based interview: **Student Response:** "this just swaps the two numbers, which ... hopefully is what it does because that's what it says it does" and **Interpretation:** Student trusts the name of the function "swap" and after briefly looking over the code, assumes that it does what it says it does.

Discussion: In analyzing the results, we see that many students do not have a firm grasp

on pass by value semantics. Several explanations exist for the observed student responses. Students appear to have relied on the function name and engaged in abductive reasoning – working backward from the name of the method to construct an explanation of the behavior of programming language constructs.

Another explanation for some of the responses is that students have recently learned about the use of a third, temporary variable in swapping the values of two variables of interest and that this example of recently acquired knowledge attracted their attention to the detriment of attention paid to parameter passing (i.e., cognitive load). Their knowledge is *fragile* – students answer some questions about pass by value correctly, but then fail to do so on other questions. Perhaps it is the presence of distractions that causes them to lose their grasp of that understanding. These distractions include function names that indicate pass by reference semantics (swap, subtract, etc.) or complex or recently learned features such as the swap behavior.

Students appeared to be attempting to make sense of code that did not actually perform a useful function in that it did not return a value nor did it alter the variables in the main method corresponding to the parameters. Testing student knowledge under such conditions (intentionally misleading names and non-useful functionality) differs markedly from the conditions under which they learned about programming, which likely used a plan-like approach[188] versus a syntactic approach[139].

Although these theories may explain some cases, the reasoning students gave indicates that a number of students do not fully grasp the concept of PBV functions, even in a fragile sense. Given these misconceptions and fragile knowledge, instructors may wish to further explore alternative ways to present the concept of pass by value, beginning with a plan-like approach to build understanding in a realistic context and with the use of visual, concrete examples. In addition, they may wish to supplement that approach with "puzzler" questions such as those used in the survey to expand students' detailed understanding of the semantics of language features such as parameter passing and to provide students with opportunities to push the boundaries of their thinking around this topic.

In studying student conceptions and misconceptions about parameter passing, we plan to engage in "live coding" interviews, in which students are given a short problem description and asked to construct a solution while thinking aloud. This context should allow us to tease apart whether misconceptions about pass by value semantics already exist in students' minds or if these apparent

misconceptions are introduced via abductive reasoning during the survey.

4.4.2 Global Variable (GlX)

A global variable is a variable declared outside of all functions. It is visible in all other functions and blocks unless a local variable of the same name hides the global variable. (i.e. "variable shadowing"). The GlX code indicates that students had the misconception that global variables cannot not be accessed from the main function. These misconceptions appeared in Q18.

Student Response: "Because answer = 7 isn't in the main scope" and **Interpretation:** Student thinks that the global variable is not in scope in main, which leads them to believe main cannot access it.

Student Response: "Line 1 is outside of either function" and **Interpretation:** Student mentions line 1 being outside of either function. This would imply that the global variable is not in scope in either function.

Student Response: "Line 1 has no affect on answer" and **Interpretation:** Student directly mentions the global variable not having an effect on the final answer.

Discussion: There were multiple cases of students not realizing or believing that global variables can be accessed from within the main function. This misunderstanding of how global variables operate was surprisingly common, but was not an anticipated misconception. However, that several students mentioned the lack of a **const** modifier is a clue to the origin of this line of reasoning. One explanation is that students have only seen global variables used in practice as a global constant, and would not have seen an example of such variables being modified. The student responses show a lack of understanding of the functionality of global variables, but their comments support the notion that students are learning in a plan-based versus syntax-based way. We are conducting further task-based interviews that we hope will shed additional light on student reasoning on this topic and on interventions that can be taken to expand student reasoning about access to global variables.

4.4.3 Overwrite "False-sharing" (OW)

This is a specific type of misconception about scope in which students believe that a variable with the same name but in a different scope can be modified by a write to the local variable. These

surfaced in the context of Q14.

Student Response: "The area of the shape is 32. The main function passes the values of x and y to the function findArea. findArea is called with the input parameters of x and y, which = 4 and 8, respectively. Area = 4 * 8 because it is written in the findArea function" and **Interpretation:** The student believes that when the findArea function assigns a value to the local variable area, that the area variable in main is also updated.

Student Response: "32. The integer area is returned but I think the compiler won't like that area is being defined at 2 separate times during execution (not sure how that will impact the program)" and **Interpretation:** Mentions that the integer area is returned even though the findArea function is void and returns nothing. The student also is aware that area is being defined two separate times, but based their answer on the local definition to the findArea function.

Student Response: "The area of the shape is 32. The main function passes the values of x and y to the function findArea. findArea then gives "area" the value of length and width (x * y or 4 * 8)" and **Interpretation:** This student only mentions an area variable once. This could mean that they did not notice the local area defined in main or that they believe that both areas are the same and the findArea one will overwrite the one in main during execution.

Discussion: Misconceptions about false sharing occurred in questions where same-named variables were declared both in main in another function. Similarly to Global Overwrite, this misconception suggests a lack of understanding of scoping rules and confusion about the existence of two separate memory locations, despite the same names. We believe that students are thinking that variables with the same name but different scopes are in fact the same variable. It is not clear whether students have a strong mental model that supports this belief, or if they are merely foraging for a reasonable explanation of some code that does not operate as expected from the naming scheme and structure seen (i.e., engaging in abductive reasoning). Ongoing task-based interviews may help us to clarify what is happening. Instructors may want to develop interventions that begin with straightforward applications of plan-based instruction and then move on to "puzzler" examples such as this one, exploration of which will allow students to construct more elaborate knowledge structures about memory models and scope.

4.4.4 Global Overwrite "Variable Shadowing" (GlOW)

This is a more specific type of OW misconception, in which students had the belief that a global variable could be modified by writing to a local variable of the same name. These misconceptions were expressed in the context of Q18.

Student Response: "Answer is now a global variable that all functions can access. Therefore, the result of subtract() can be passed back" and **Interpretation:** Student appears to believe that since answer is a global variable, the result of a local function would be returned into the global answer variable as opposed to being passed back to main as a return value.

Student Response: "because answer is a global variable, the subtract function will change the value" and **Interpretation:** Student believes that the subtract function will be able to alter the value of the global variable answer.

Student Response: "Answer was not a const, so after 5 - 8, answer is returned as -3. Since the global variable answer is not a constant, the value of it can be modified, and it is just before the print statement." and **Interpretation:** Student believes that global variables must be declared as constant in order for them to not be overwritten by a local variable of the same name.

Task-based interviews were conducted for Q18, with the following responses: **Student Response:** "I think you would get a warning on that because it doesn't like it when you have a global and a local variable of the same name" and **Interpretation:** Student, after going through the code line by line, has difficulty knowing how the idea of variable shadowing, having a global and local variable of the same name, would be handled.

Student Response: "this gets reinitialized up at the top from 7 back to -3 . . . Because that's a global variable but it's not immutable. I mean, it can be changed." and **Interpretation:** Student correctly believes that a global variable's value can be changed. However, they believe that a local variable of the same name can modify the global variable.

Discussion: This misconception was seen when students thought that global variables were modified by writing to a variable of the same name. Some students thought the global declaration needed to be type **const** to avoid this while others just thought the variable could be modified in this way. Students in this course have typically seen global variables used as constants and except for a few examples of local versus global variables in a class exercise, have not made use of global variables in their programs except as read-only, global constants. The underlying issues here seem

to be both a lack of understanding of *variable shadowing* (thinking that variables of the same name are the same variable) and lack of knowledge of how return values are passed back to calling code.

The structure of this course shows students the idea of defining a variable and then using that variable in multiple statements, all of which refer to that same original variable. Students have also seen examples of variable shadowing in the use of same-named variables in the main function and in other functions. Again, however, their grasp of this concept is fragile and they appear to construct explanations that fit a surface interpretation of the code's purpose.

To build robust knowledge about scoping, students would have to be exposed to situations in which they experience unexpected outputs when they attempt to associate uses of variables with definitions that occur in different scopes. Instructor knowledge of this gap in understanding would allow instructors to design appropriate in-class examples or to incorporate associated elements into projects that would help students to solidify their understanding of this concept. Again, we plan to explore student thinking around this topic more thoroughly via task-based interviews, using both paper-and-pencil examples and live-coding.

We report several types of misconceptions observed in students who have just completed a C-based CS1 course at a large public university that specializes in science and engineering. These misconceptions center largely on parameter passing and scope.

4.5 Conclusions

Though the existence of these or similar misconceptions has been reported by others, the detailed explanations provided by students provide some insight into why they may think this way, and can provide instructors with a basis for crafting high-quality instructional materials that stimulate students to expand their knowledge structures to include more detailed views of the related language features and structures. Other studies have determined common mistakes or misconceptions through eliciting responses from teaching assistants, professors, or solely examining student responses to exam questions [91, 23]. We have worked on a process that allows student thinking to drive the results, and the various stages of the process provide a form of verification to help eliminate speculation.

The work that we have done gives us a starting point to further examine these concepts. The misconceptions described in this paper may be of interest for work in threshold concepts or

fundamental ideas in imperative programming languages. More testing can be done to see how these concepts fit into the categories of threshold concepts (transformative, irreversible, integrative, etc.)[134]. It seems clear that these concepts are troublesome for students, but further detailed comparison to related work will allow us to better classify them.

We explore several approaches to extracting details of student reasoning: the pencil-and-paper "explain your reasoning" type questions in the survey, task-based interviews of students engaged in answering such questions (ongoing) and task-based interviews of students engaged in live coding. This live-coding will differ from work done by Craig[43] and Cherenkova[28] by having the students supervised and in that it will focus more on the qualitative data that we can gain from a think-aloud protocol.

A key limitation of this work is that it focused on one semester of one class at a single university. Additionally, this university differs from many in that the CS1 course focuses on the C language, and is thus not object-oriented. Further, the work we present here focuses on a subset of four questions rather than a full analysis of all of the questions in the survey, which in turn are a selected subset of the concepts taught in an introductory CS course.

Other potential limitations include the nature and the wording of the questions in the survey. Asking students "What is the *result* of the execution of this code?" while having variables such as *answer* and *solution* could have confused some students. With the questions themselves, we want to ensure that we are uncovering genuine misconceptions and not prompting students to form misconceptions with code snippets that appear to do one thing but do not behave as expected. Some of the questions might trick students because the code snippets contain poor coding practices that instructors do not teach. As mentioned above, the questions are "puzzlers" in that they are designed to separate the actual behavior of the code from the apparent purpose and context. This is not how students typically experience code, and so it is not clear if the expressed misconceptions are firmly held or if they are the result of abductive reasoning or information foraging behavior, which is typical of debugging behavior. We would like to focus in future work on more high-level concepts and will remove some syntax-focused questions such as modulus operators, order of operations, the difference between pre and post-incrementer, etc.

The goal of our work is to give a more concrete understanding of not only misconceptions that students have in introductory computer science classes, but also why they are having them and longer term, to develop and evaluate interventions to address these misconceptions. In this way,

we hope to aid in the creation and evaluation of high quality pedagogical materials for computer science.

Chapter 5

Coding in the Wild

This paper was accepted and will appear at the 2019 Innovation and Technology in Computer Science Education conference held in Aberdeen, Scotland under the title "Qualitative Observations of Student Reasoning: Coding in the Wild." [105]

5.1 Introduction

To obtain knowledge about what students are thinking and which concepts they find difficult, some researchers have developed concept inventories [23, 201, 148, 86] or conceptual assessments [132, 123, 104], and others have utilized task-based interviews to gain insight into student reasoning about programming [104, 90, 79]. Concept inventories typically provide short code examples and gauge student reasoning based on their answers to multiple choice questions. Other assessments look at the outcomes of student efforts to create programs that solve particular problems, but do not follow their thought processes as they engage in the problem-solving task.

In this work, we seek to answer the research question: *How do intro CS students reason about the design and implementation of simple programs* ***in the wild****?* By "in the wild" we refer to coding that occurs without any initial skeleton or detailed instruction on how to approach the programming task. We seek to study how students reason as they are engaged in reverse engineering a solution from scratch, which may differ from their reasoning processes under the conditions associated with the administration of concept inventories. In the context of our study, students were not provided with any "starter" code but rather with an executable and were asked to recreate the functionality

of this executable from scratch. We did provide them with two simple constraints, described in section 5.3, designed to force parameter passing. Thus, the coding sessions may be more precisely described as "in the semi-wild."

We conducted a think aloud study[141, 61] of introductory CS students in a course using the C programming language at a large, public US institution. The student programming session was captured on video, and their timestamped actions and utterances coded using thematic analysis, a method for identifying, analyzing, and reporting patterns or themes in qualitative data[15].

We report on the students' approach to implementation, the CS concepts they find troublesome, their uncertainties and resolutions (or lack thereof) to those uncertainties.

5.2 Background and Related Work

Prior work observed student reasoning by having students analyze code or attempted to gauge misconceptions based on student responses to conceptual questions. This work includes concept inventories, criterion-referenced tests designed to help determine a student's knowledge of a specific set of concepts and to expose misconceptions. For example, Tew and Guzdial[201] developed the FCS1, a validated concept inventory that uses pseudocode in its questions, and Parker et al.[148] replicated the inventory as the SCS1 to enable broader distribution of the tool.

Other researchers developed assessments of programming ability and knowledge of CS concepts [132, 54, 104]. McCracken et al. [132] identified learning objectives for novice programmers and found that first year students perform significantly worse than expected when considering these objectives. We consider these objectives in our thematic analysis. Drachova [54] developed an inventory to measure principles for reasoning about correctness of software.

In a previous study, we developed a CS conceptual assessment that probed further than concept inventories by asking students to "Please explain your reasoning" after each question[104]. This assessment was meant to gauge misconceptions on concepts that Goldman et al.[78] had identified as "important and difficult." We categorized these misconceptions and found that students struggled with pass-by-reference versus pass-by-value semantics, "false sharing" of variables with the same name but in different scopes, and the semantics of access to global variables[104]. To further support these claims, we followed up with one-on-one task-based interviews requiring students to explain code snippets and allowing them to clarify their thoughts in a way not always possible

through researchers retroactively reviewing responses to assessment questions.

In this work, we seek to analyze students' thoughts and actions as they try to complete a programming task. This method of obtaining student reasoning is similar to that of Craig [43] and Cherenkova [28], who both analyzed student code samples to gauge student reasoning. Cherenkova [28] did this by gathering over 250,000 student responses to weekly code-writing problems with the goal of identifying concepts that students found challenging. Her work showed that students have significant difficulties with conditionals and loops that persist throughout the duration of a course.

Craig's work looked specifically at the concept of pointers[43]. The data collected and analyzed were a mix of responses to multiple-choice questions and submissions of coding exercises based on pointers. Over 300 students submitted artifacts and results showed that students "confuse an address with a pointer," have trouble understanding the relationship between pointers and arrays, do not have a robust mental model of assignment statements even in their second year, and might apply operators blindly in an attempt to make the types consistent in their code. Craig goes on to say "while our analysis suggests that students misapply specific structures, a different type of study, one featuring think-aloud solutions to problems, for example, will be required to determine what mental models students are using in these situations." This motivating quote is precisely what our study aims to accomplish.

5.3 Experimental Design

We developed a simple calculator program designed to exercise selected concepts from CS1. The output of one iteration of the program is seen in figure 5.1, and an abridged version of an example correct implementation in figure 5.2. The process shown in figure 5.1 repeats until the user enters an exit code at this prompt (number that is not 1), and the final output gives the total number of operations performed.

5.3.1 Study

We solicited participants from CS1 classes near the end of the semester and from CS2 classes at the beginning of the semester. The researchers followed all policies and procedures of the university's Institutional Review Board and appropriate approvals were obtained for the study. Participants were incentivized to participate with $10 Amazon gift cards. One researcher met indi-

Figure 5.1: The example calculator program

```
Please select an operation to perform:
1 = addition
2 = subtraction
3 = flipping the signs
2

Please enter an integer:
7
Please enter another integer:
16

The difference between 7 and 16 is -9.
```

vidually with participants, briefly explained procedures and provided participants with an instruction sheet containing two rules for successfully completing the task: 1) The three operations (addition, subtraction, and flipping the signs) must be implemented in separate functions. 2) The statements that print out the results of the operations must be in the main function. These rules were meant to ensure that their solutions did not avoid the usage of key topics such as functions, return values, and pass by value/reference semantics.

The participants completed a demographic survey online before beginning the coding task. All of the participants were confident or very confident in their CS course materials (Agree or Strongly Agree on Likert-Scale questions). We used the think-aloud method[141, 61] with the goal of obtaining a deeper understanding of what students are thinking as they reason about problem solving and software development in this context. The think-aloud method has been widely used in Human Computer Interaction research [141]. As recommended, we periodically remind participants to "Please keep talking." [60]. We used a time period of 120 seconds before prompting. Participants were given 60 minutes to complete the task.

5.3.2 Demographics

The ten participants ranged in age from 18 to 25; eight were male and two were female. Although CS1/2 are introductory level courses for computing majors, they are also taken by non-majors. Eight of the ten were working toward a BS/BA in Computer Science or a BS in Computer Information Systems, one was working toward a CS minor and one was a Mathematical Sciences BS student. Three classified themselves as freshmen, four as sophomores, two as juniors, and one as a

Figure 5.2: An example solution, abridged

```
static int count = 0;
int add (int x, int y) { return x + y; }
int subtract (int x, int y) {return x - y;}
void flipsign (int num[2]) {
num[0] = num[0] * -1;
num[1] = num[1] * -1;
}

int main() {
int go, one, two, operation, answer, array[2];
go = 1;
.... display menu and obtain initial operation code

while (go == 1) {
... solicit and obtain input for each operand
if (operation == 1) { answer = add(one, two); ... and print result }
if (operation == 2) { answer = subtract(one, two); .. and print result }
if (operation == 3) {
array[0] = one; array[1] = two; ... display original values
flipsign(array); ... display updated values in array[0] and array[1]);
}
count++;

// ... get another operation code or stop code
if (go != 1) { // ... print number ops performed }
}
else { ... again display menu and get operation code }
}
```

senior.

5.3.3 Preparing the Data

The researchers reviewed the videos and transcribed the audio into timestamped, segmented "utterances" in a spreadsheet. Interesting phenomena were noted and researchers compared their notes across participants to develop an initial coding scheme that captured their observations.

After further discussion including a third researcher, it became apparent that the coding scheme could usefully employ codes associated with well-defined concepts in the literature. The authors both coded each of the remaining transcripts individually using this scheme, and came together to discuss and come to consensus on the coding, with a few additional sub-codes emerging from this process. A third researcher reviewed a subset of the videos, providing cross validation for the work. The final categories for which we recorded data were: **Time**, **Utterance**, **Action Code**,

Current Task, **Problem Solving Phase**, **CS Concept**, and **Proposed Certainty Level**. Those with specialized codes are described below:

Action Code: The potential actions were Compile, Save, Edit, Execute Own, Check Executable, View, Review, Tools, or Comment.

Current Task: The potential current tasks were Main (Menu), Main (Loop), Main (Counting Operations), Addition function, Subtraction function, or Flipping Signs function.

Problem Solving Phase: Based on McCracken's[132] framework for the learning objectives of novice programmers, we used the phases Understanding the Problem, Breaking Into Subproblems, Implementing Solution (Subproblem), and Implementing Solution (Combine).

CS Concept: The codes we used for this work were based on Goldman[78] and included Parameters/Arguments I/II/III (PA1/2/3), Procedures/Functions/Methods (PROC), Control Flow (CF), Types (TYP), Boolean Logic (BL), Syntax vs. Semantics, Operator Precedence, Assignment Statements (SVS), Scope (SCO), Abstraction / Pattern Recognition and Use (APR), Iterations/Loops 0/2 (IT0/2), Arrays I/II/III (AR1/2/3), Memory Model, References, or Pointers (MMR), Design and Problem Solving I/II (DPS1/2), Debugging / Exception Handling (DEH), and Other (OTH). The Iterations/Loops 0 category was created by the authors using the same description as Iterations/Loop I, but replacing "nested loops" with "non-nested loops."

Proposed Certainty Level: The codes for this column were No Knowledge, Uncertain, Muddled, Certain (Correct), and Certain (Incorrect). These were used to categorize our view of participants' confidence in what they did/said.

5.4 Results

As described above, each of the authors individually reviewed all 10 videos and then engaged in a discussion to reach consensus on any points of difference in their coding.

Evaluating Success. The overall results of the study with respect to success at recreating the functionality are: **Successful**: Seven participants (4 CS1, 3 CS2) and **Not Successful**: Three participants (1 CS1, 2 CS2). Our definition of success did not hinge on having exact wording or spacing for the output prompts or results. Rather, we defined a successful implementation as one that showed the menu, accepted a menu choice and two integers and printed the result, all in a loop that continued until the user chose to exit, and then finally displayed the total number of calculations

performed.

Qualitative Analysis. Our primary results are qualitative in nature. This section reports on the observations we made about students' solution approaches, issues and resolutions, and their perceived self-efficacy. Solution approach captures the order in which the students worked, the order of elements in the program, any observable guiding principles to their workflow, when and how often they compiled and executed, whether they stopped to review and reflect on their code or engage in any refactoring or code cleanup, as well as any indications of planning, avoidance or delay. They typically implemented the elements of the program in the order in which they encountered the functionality in the provided executable, beginning with the initial menu prompts in the main method, and adding variables needed to store operation codes, operands and results.

The issues and resolution sections capture aspects of the program about which participants were uncertain or exhibited misconceptions and describes if and how they were able to resolve the issue. In terms of self-efficacy, most students exhibited some form of self-talk, often critical, which we capture and provide evidence of in the following sections.

Due to space limitations, we report in detail on only five of the ten total participants of the study, chosen because their results taken together provide coverage of the key insights gained from the study. However, we included our summary of all 10 participants in the discussion section.

5.4.1 Participant 0049 (CS2)

Solution Approach

Participant 0049 followed the common solution approach described above and it was clear that his development process was driven by the behavior of the provided executable. The factors guiding his progress were feedback from the editor, from the compiler, and from the example executable and he was ultimately successful.

Issues and Resolution

Participant 0049 encountered a variety of interesting issues, many of which he was able to resolve and some that he remained uncertain about and decided to ignore.

Issue: Omissions and typographical errors; **Resolution:** Editor feedback, compiler messages and code review. This participant noted that changes in the color of text in the editor indicated some syntax errors.

Issue: function parameters (PROC); **Resolution:** code review, recall. The participant

implemented the addition function but was uncertain if he needed to actually name the formal paramters or if could just use the types. He later reviewed his code and realized that he needed to use names as well as types.

Issue: call-by-reference, parameters, return values (PROC); **Resolution:** compiler messages, testing, not completely resolved. In implementing the flipping method, his thinking was muddled. He realized that something different was needed here, saying "when flipping a sign, you're going to need to return two values, so ... there are a few ways you could do it: declare a "regular pointer" and then return that pointer ... returning an integer array that will hold 2 values." His interesting solution was to create a flipping function with three input parameters: an integer intended to hold a copy of the first element of the array, an integer to hold a copy of the second element of the array, and then the array itself. The function attempted to return the array, although the function return type was simply an integer.

His series of edits were largely guided by compiler error messages: he was "99% certain" that he should not have to place a '*' in front of the array name to return the array, but he did so anyway based on compiler messages (he was returning an integer array but had declared a return type of int). This version compiled and his output was correct because he assigned the flipped values to the array elements before his return statement "return *newVals," which actually returned a copy of the value of the first element of the array. However, his simple tests passed and he was satisfied with the final product, despite his unease with dereferencing the array name in the return statement.

Issue: function prototypes (TYP); **Resolution:** compiler errors. Compiler errors revealed that his placement of the addition, subtraction, and flipping function after the main method meant that he had to declare prototypes prior to the main method.

Self-efficacy

Participant 0049 was confident and appeared quite comfortable with the assigned task and exhibited little to no self-critical talk. Rather, he exhibited curiosity, with statements such as "Let's just see" and "I'm pretty sure that ...".

5.4.2 Participant 0048 (CS2)

Solution Approach

Participant 0048 took the standard approach of running the executable and then breaking the task into parts, tackling them in the order in which they appeared in the menu of the provided executable.

His programming decisions were driven by classroom experience, having learned from mistakes or feedback received from previous classes. He worked through trial-and-error in multiple instances to solve problems, and expressed that he was aware of the fact that this method helped with and was part of the learning process. He compiled throughout the task and used the compiler errors to resolve issues. He also showed signs of planning ahead when programming, speaking of "wondering also how to flip these signs" while working on earlier parts of the program. These approaches allowed him to successfully complete the task in the allotted time.

Issues and Resolutions

Participant 0048 was able to successfully interact with the compiler feedback to resolve many problems.

Issue: Function prototypes and return values (PROC); **Resolution:** Compiler messages, recall. On one compiling instance, he notes "Oh, cause I don't have function prototypes...And then I need to make sure I return 0 because that did throw me a warning." In instances where the participant isn't clear on the compiler error given, he has ideas of potential resolutions to problems. A great example of this are the utterances "I'm just not incredibly sure what all of that means, but I'll figure it out." "Should I use...is it like star or am I gonna have to do the &? I'll find out in a bit." This particular grouping of utterances was dealing with pointer functionality (Pointers).

Issue: Pointers (MMR); **Resolution:** Compiler messages, trial-and-error. The concept of pointers were where participant 0048 had the most issues. He implemented the flip signs function using them, but he spent time going between compiler errors and trying to fix the syntax of the function. Although the participant had enough knowledge of how to implement pointers that he was successful, he still exhibited muddled thinking regarding the semantics of referencing and dereferencing pointers, such as when to use "&" versus "*."

Something that stood out about this participant was his explicit mention of learning occurring through working on problems and getting feedback. He says "That's why I don't do good on exams because I like to test my programs. I guess a lot of how I do it is throwing things at it and seeing if it works..just because it helps me learn what I should do and what I shouldn't." This speaks to the use of written assessments in computer science, noting that students who may not be able to give a correct exam answer about a concept may be able to demonstrate correct use of that concept when asked to implement it. Another example of this learning-focused language occurs when he says "So I'm actually going to see if that will even compile. Cause that's where I fix a lot

of my stuff." These utterances show evidence of the benefits that can be gained in CS from working at problem solving and constructing knowledge using a combination of classroom experiences and information with hands-on practice.

Self-efficacy

Participant 0048 had multiple occurrences of using self-deprecating language in a light-hearted manner, once stating "Oh...I'm *dumb*..almost didn't catch myself" when referring to declaring a variable without giving it a type (TYP) and once stating "So I've been wondering also how to flip these signs and maybe that's *dumb* that it took me this long but, if I just multiply the values by -1 then they'll flip...obviously." The other example of self-efficacy directly speaks towards confidence, with him saying "Great. I'm confident that I can" referring to being able to successfully complete the task.

5.4.3 Participant 0043 (CS1)

Solution Approach

Participant 0043 followed the standard approach of running the executable and then working on tasks in the order seen in the executable. He resolved problems through either reviewing his code or deciphering compiler feedback to successfully complete the task. He did not compile many times (5), but did check the executable many times to clarify formatting and functionality, allowing him to successfully complete the task.

Issues and Resolutions

Issue: Array declaration (AR3 + TYP); **Resolution:** Compiler messages, code review. Participant 0043 encountered an error with variable declaration. Alerted to the issue via a compiler message, he commented "Conflicting types..Oh. Probably just because you can't do the logic while you're declaring it," but he soon realized, "Ah! I declared it twice" after going back to look at the code. He was able to resolve those issues through compiler feedback.

Issue: call-by-reference, passing array as argument (PA1 + AR2); **Resolution:** avoidance. While working with the flip signs function, he originally attempted to declare and return an array. He received a compiler message, and said "Invalid initializer. Did I declare up here? Nope! That'll do it," showing the ability to understand the compiler error message and remedy the mistake. He remarks that "they told us how to do it but I can't remember" with respect to returning two variables in a function but then "maybe that's not how they told us to do it" after running into compiler

errors with implementation. The resolution eventually comes in a problem solving epiphany of just calling the flip signs function twice, allowing it to just return one value. This workaround allows him to "avoid the problem entirely."

Self-efficacy

Participant 0043 used language that might imply low self-efficacy, but said it in a light-hearted manner that made it apparent that he did not actually lack confidence. When catching a minor syntax error, he remarks "Whoops. I'm *stupid*." Also, when implementing addition function, he says "So sum's gonna equal num1 + num2. Easy!" showing confidence in working through the task.

5.4.4 Participant 0564 (CS2)

Solution Approach

Participant 0564 followed the common approach of running the executable and then working on tasks in the order seen in the executable. He did not compile for the first time until about 44 minutes into the study. This led to an overwhelming number of errors and issues to fix within the allotted time. He eventually attempted to avoid the issues faced when too many errors arose. An interesting facet of his solution approach is that he did check for extra input validation by the executable, stating "Well I tested this to see if it would produce an error message if I used a number that wasn't 1, 2, or 3." This participant was not successful at completing the task in the allotted time.

Issues and Resolutions

Issue: call-by-value vs. call-by-reference (PA1); **Resolution:** not resolved. This participant encountered difficulty with parameter passing. However, his decision to delay compiling until quite late in the session prevented him from locating the error. He attempted to resolve multiple errors from the feedback of the compiler one at a time, noting "End of non-void function..." "z undeclared...Ok. e undeclared." "Expected before..e and z...Line 56."

Issue: Scope (SCO); **Resolution:** not resolved. This participant encountered scope (SCO) issues in which he attempted to access a variable local to a function while in the main function. He then decided to work on other parts after encountering some errors, such as when stating "Error label at end of..Line 63..? Why am I getting this error?" referring to the switch statement he had attempted to implement. Scope issues were prevalent. He spent 44 minutes editing his code and running the provided executable before attempting to verify the logic of the three functions, and mostly understood how to successfully do it. He used an interesting algorithm for flipping signs (x =

x - 2x). However, not being able to resolve the issue of capturing the return value from the function, he was never able to successfully compile or execute his code.

Self-efficacy

0564 used the word "tripping" multiple times to refer to making a mistake during the task. After seeing a compiler error, he remarked "Dang. I'm tripping." Another error led him to say "Why am I getting this error? I'm tripping. What am I doing?" These utterances indicate uncertainty, but the participant seems to feel the issues should not be occurring.

5.4.5 Participant 0044 (CS1)

Solution Approach

Participant 0044 followed the common approach described above of running the executable and then working on sub-tasks in the order encountered when running the executable. She showed signs of avoiding issues and moving to work on other parts if resolutions could not be found. She compiled early and and frequently (16 times), but was unable to resolve her issues and ran out of time before she was able to display the result of either of those functions and was ultimately unsuccessful.

Issues and Resolutions

This participant was able to resolve syntax issues through the help of compiler feedback. Commenting things such as "Oh! I didn't put the function header. Yeah. That would do it" or "Parameter names..without types. Awww. I forgot that. Yeah. I would." After implementing the addition function below the main method with the function prototype in the main method, compiling and receiving errors, she was able to resolve this issue.

Issue: Parameter passing (PROC) ; Participant 0044 encountered issues with function prototypes returning values from a function and with receiving the result of function call in the calling context (PROC). She attempted many variations of the code but was unable to resolve the issue of the combined action of returning a value from the function and capturing that returned value in order to print it (PROC).

Issue: Scope (SCO) ; **Resolution: not resolved** . She was confused about scoping rules (SCO) and seemed to believe that having variables in the function with the same names as in the main method would result in a transfer of the values. Her combined confusion about scope and return values led to consternation on her part and prevented her from successfully completing the assignment. She tried multiple ways, but continued to get values of 0 regardless of the integers

input when executing the program. "So it's still 0. Hmmm. Why is it?...I guess I should just try it a different way." Eventually, the participant moved on and attempted to work on other parts of the task, uttering things such as "Wait. Work on something else and then come back to the.." and "Ok. Let's move on...Try something else."

Self-Efficacy

This participant showed multiple instances of low self-efficacy, doubting her own skills and abilities. One common theme was the feeling of "Ugh. I'm making this really complicated.," She said this or a slight variation of this seven times. Participant 0044 also felt that the struggle was unnecessary, remarking "I feel like this is really easy and I'm overthinking it" and "Ugh. This should not be this hard."

5.5 Discussion

We observed student uncertainty about scope, pointers, arrays, function prototypes, parameters and parameter passing, and return values. These students exhibited fragile knowledge, a result that was found in our previous work looking at conceptual assessments [104]. Students were able to resolve their uncertainties in various ways: editor feedback, compiler error messages, reflection/recall, code review, and testing, but these resolutions did not always equate to a full understanding of the concepts. Although students relied on compiler error messages to resolve uncertainties and to uncover issues, they had difficulty interpreting compiler error messages, a problem Gusukuma [80] noticed and worked towards remedying through the use of misconception-driven feedback.

We also observed that the ability to create a program with correct output did not equate to full comprehension of the programming language concepts and features employed. For example, we saw students who could not properly implement call-by-reference succeed in creating a "correct" program output but only through trial-and-error and workarounds that they were certain were incorrect. Such workarounds and trial-and-error were previously reported by Hundhausen [95] and the goal of moving away from trial-and-error in student programming is discussed by Edwards [58].

We found pointer-related evidence to support the result of Craig's [43] work claiming that students apply operators blindly in an attempt to get their code working. From a design perspective, we note trends of students' decisions with respect to implementation order. Most participants

attempted to implement the program in the order that the executable was presented, starting with the main method (menu) and then working on the addition, subtraction, and flip sign functions in that order, and lastly implementing the loop and counter functionalities. This caused some students to run into errors as they did not originally include function prototypes before the main method. Multiple students mentioned working on the flip signs function last because it seemed trickier. We would be interested to explore the effect of changing the order of the functions in the executable, placing the flip signs first, to see if students would follow the observed order or first tackle the easier functions. Students showed signs of avoiding complexity, sometimes devising innovative solutions to solve a problem. For the flipSign function, we saw solutions such as having the function return one value but calling it twice in main (0045), attempting to "put a dash in front" of the number to flip the sign (0045), and using number line thinking for the algorithm to flip the sign as opposed to multiplying by -1 (0042, 0043, 0045, 0564, 0045).

This work allowed us to view unexplored features of student reasoning such as the self-efficacy of the students, which gave insights into level of certainty, their problem solving approaches, and how they resolve issues. Students show evolved reasoning about the semantics of programming language constructs as they interacted with the edit-review-compile-test cycle. Practice such as this helps students to learn. However, we saw that these feedback mechanisms alone may be insufficient to uncover student misconceptions about such semantics, or worse yet to induce flawed mental models of the actual semantics, which means that evaluating student knowledge based on correct output alone may lead to the introduction and/or "hardening" of misconceptions (0049). In summary, intro CS students while coding "in the wild" seem to follow a few trends: focus on the order of the specification provided; avoid complexities through workarounds or innovative solutions that sometimes work but may not address the intended concept instructors would like students to learn; and use compiler feedback combined with the fragile knowledge of concepts to address problems through trial-and-error.

This work suggests several pedagogical techniques: carefully crafting programming assignments to force correct usage; evaluating code and not just correct output for programming assignments as suggested by Edwards [58]; mining studies like this one and student submissions to find dysfunctional code that "works" but embodies misconceptions and using these as instructional tools in which students are asked what the code does or why it coincidentally works; and incorporating the knowledge gained into CS pedagogical content knowledge, allowing instructors to create assignments

and assessments that probe for these uncertainties and misconceptions in the local context.

We also observed problem-solving approaches that promoted student success, such as intermittent code review and incremental development. When students encountered uncertainty, they either delayed implementation of the problematic feature (0044), avoided it entirely (0564) or engineered a solution through trial-and-error that worked, but that they did not understand (0565, 0049, 0048).

Limitations of the work may include generalizing from our context: our participants came from a CS1 course at a large, public university in the United States. Some of the participants had just finished CS1 in December. Others were just beginning CS2 but in August, so they had a summer during which to "ripen," though they had no additional formal instruction. Different phenomena may be observed in students at other stages of their CS educations or with more or less prior preparation. Reaching out to classrooms and offering incentives for participation is not uncommon as a methodology, but it is dependent on students' willingness to provide their time, especially in a one-on-one research setting. Thus, the students in this study may be representative of those who are willing to volunteer their time for outside of class CS work. Students also were programming in an editor not of their choosing. To ensure consistency, we had all students use the same editor for the study, though it turned out that this was not the preferred editor for any of the students. In the future, we would like to study students at different levels of their CS education, engaged in implementing programs that exercise a wider variety of CS concepts.

5.6 Acknowledgements

The authors of this paper acknowledge Aubrey Lawson for her contributions to the coding scheme and data analysis, suggestions for related work, and insights included in the discussion section.

Chapter 6

Software Engineering Design Documentation Interviews

6.1 Introduction

Collecting and analyzing records of students' actions, speech, and writing while they are engaged in software analysis, design and implementation can provide important insights into how students think about key concepts and practices in computer science. These insights have the potential to influence the design of improved CS pedagogy. In this paper, we focus on student thinking about software design and specification in an object-oriented (OO) context.

Some debate exists in the CS community as to whether the OO or procedural paradigm between OO and procedural is more difficult to teach [124, 174]. To begin to tackle this problem, research has examined misconceptions relating to OO programming [74, 88, 204, 56, 66, 167, 50]. We seek to understand the nature of how students inexperienced with design and specification think about and engage with the process of design and specification of software artifacts in an upper-level, software engineering course. We collected documentation from 8 teams at a large, engineering-based public US university and interviewed the groups based on the documentation provided. The students were enrolled in a course that is both project and team based, many for the first time. The documentation consisted of UML diagrams, descriptions of use case scenarios, and prototypes user interface designs. All of these artifacts are meant to be consistent with one another to ensure

proper design. We take an exploratory approach, asking open-ended interview questions focused on but comprehending the thought process that the students have with respect to OO design, following a requirements analysis that they had performed. We compare their statements about their designs with the actual artifacts they created, looking for emerging patterns and themes. In the following sections, we provide a detailed look at the related work in computing education research surrounding OO design and analysis, describe the methodology for our own study, present the results for each student team, and discuss these results and the limitations.

6.2 Background and Related Work

Much prior research has focused on object-oriented (OO) programming and design.

Flores et al.[67] examined the principle of information hiding [150, 149] through a qualitative study. This study had software design students in Europe read over an information hiding article by Parnas [150], have a two hour discussion on it, and descriptively design a software application with the information hiding principle in mind. The researchers then reviewed the assignments collected, conducted open interviews with the students, and then tutored them. Data collected included audio recordings and documents with diagrams and descriptions. The audio data was collected for the analysis and discussion of the Parnas article and during the tutorials and open interview with the students. The study found that none of the students had a clear grasp on the information hiding principle and classified how students seem to view the principle into two main ideas: 1. Hiding information consists of hiding what is inside modules, regardless of the form of division into modules, and 2. The division produces information hiding. In this study, we follow a similar methodology as [67], also analyzing documents and interviewing students. A key difference is in our exploratory and open-ended approach to the interviews. Compared to Flores et al., who used questions geared towards the understanding of a specific principle, our work looks to discover the conceptions that students have related to their design of an OO system, comparing both the context of the document with student verbal descriptions and comparing different representations.

Ragonis and Ben-Ari [167] found useful implications and suggestions to pedagogy through a two-year study of novices' comprehension of OOP concepts. They originally attempted to avoid teaching the novices about the main method, choosing to focus on the actual OO process and concepts such as classes and objects. They found, however, that indefinite postponement of teaching

the main method interferes with understanding the dynamic aspects of programs. After one year of their study, some of the conclusions were: Classes and objects should be introduced first using diagrams; Examples using graphics should be avoided because novice students conflate the "object" with its rendering on the screen; Problems relating to computer systems should be used when teaching OO concepts, not just "real-life" problems involving employees and animals [167]. This is a subset of the conclusions found from this first year of the study, chosen for their relevance to our work. The first and second initially seem as though they would inherently contradict one another, but the research seems to suggest that diagrams and graphics are different in the manner they can be used to teach. Graphics are more high level, and give a direct metaphor link for students to learn from (e.g., a picture of a safe to represent a "bank" class) whereas diagrams make the connection not as explicit, but allow for a better understanding of classes and objects. The third result is of interest because it goes against or at least offers an extension to some of the work conducted on metaphor learning in CS [37]. This suggestion actually is supported by Dijkstra's work calling for a ban on anthropomorphic metaphors in CS[51] . This suggestion presents a middle ground, similar to the one seemingly presented on visuals (diagrams vs. graphics), admitting to the usefulness of metaphors with respect to OOP concepts, but limiting those metaphors to problems within a computer system context and not leaving them as high level to make them everyday, real-life problems, as these problems are not always realistic in a computer system and how it must be designed. Another important quote from the Ragonis [167] study is when they say "We were not just cataloguing misconceptions, but trying to understand what made for effective learning." This was the authors way of making it clear that their interest was in evidence that showed both misconceptions as well as comprehension of the OO concepts. In this same vein, our work values the correct mental models that exist as well as the flawed ones, considering both of these facets are necessary to obtain the full learning picture, which will help guide improved teaching and effective learning.

Sanders and Thomas [174] generate a checklist for grading OO novice programs based on conceptions and misconceptions they found in their work. [174] asserts that regardless of the truth in the debate on whether procedural or OO programming is more difficult to teach [124], OO is inherently more difficult because instructors are not as familiar with the paradigm and the difficulties students face. As time passes, this concern may become less relevant. Sanders' literature review found several OO misconceptions that are interest in this study, namely **problems with abstraction** [204] and **problems with modelling** [56, 204]. Sanders' study [174] collects data through

students' programming artifacts, focused on OO programming concepts, and from these artifacts, a number of considerations for grading novice students' OO programs emerged. A general result Sanders found evidence of is the importance of ensuring that grading of programs does not merely focus on the superficial aspects of it compiling and passing test cases, but considering underlying misconceptions that might exist.

Hadar [83] has looked at the OO learning issues faced by computer scientists through the lens of a Nobel Prize-winning psychologist, Daniel Kahneman. Hadar used dual-process theory [101] to posit that when it comes to OO design and analysis, there is an inherent disconnect between how it works and how our mind perceives/would expect it to work. To come to this belief, Hadar intentionally chose to conduct a study on software engineering professionals as opposed to students, and managed to find similar misconceptions that researchers have found exist in novices. Their study [83] uses a grounded theory methodology [192, 191] with a group of software developers who practice OO analysis and design on a daily basis. Three instances of a workshop were selected for analysis with each instance consisting of three 8-hour days. They selected participants who practiced OO analysis/design daily, studied OO programming and design during their undergraduate studies, and had only worked within the OO paradigm since graduation (to avoid the potential of difficulties stemming from previous experience in procedural programming). The final batch of participants consisted of 41 software developers from seven different companies. After collecting data through written solutions from participants (mostly UML models) and observations of class and pair discussions (audio transcriptions), interviews were conducted to attempt to understand the participants' thought processes. The researchers found through preliminary analysis that the dual-process theory was appropriate for analyzing the data and continued qualitatively coding with codes developed around this theory. A general trend of the results obtained showed that even professionals immersed in OO design/analysis fall victim to misconceptions similar to those that novices do and interestingly, seem to make non-normative decisions, decisions that contradict the knowledge they have about the subject. A similar result was found with respect to introductory CS topics in [104]. The simplified explanation of the results deals with Kahneman's [101] Systems 1 and 2. System 1 (S1) is responsible for immediate, automatic thinking that solves very familiar problems whereas System 2 (S2) handles problems that require more analysis and calculations. In general, S1 is triggered first and uses heuristics or familiarity to attempt to solve a problem. An example would be most adults being asked the result of 2 + 2. The correct answer of 4 would come to mind automatically and

without any real calculations involved (S1). This same question, however, posed to a child who is just beginning to learn about counting and mathematics could be an S2 process. You might notice the child counting on her fingers two and two to get to the accurate answer. This relates to the OO design and analysis work because some of these processes for professionals seem to have become S1 so that based on the heuristics, previous experience and expertise held by them, they sometimes solve them without having the more analytical and calculating S2 working to check their work. This is not made easier when some conventions are counterintuitive in OO design/analysis, such as the concept of inheritance. Inheritance outside of programming is usually seen as transferring things such as money, and generally the person who has less *inherits* from the person who has more. In a CS context however, this is not the case, as the class that *inherits* from another class can extend it and have more functionality than the class it is inherited from.

Through using these sources of knowledge and prior research experiences, we have a solid guide to frame our study's methodology and results in the following sections.

6.3 Experimental Design

For this study, we collected student class project submissions from a software engineering course at a large, public, engineering-based US university. Students formed teams of three students, typically of their own choosing. The project had two major assignments, broken down into a series of submissions. The objective of **Assignment one (A1)** was to develop the requirements for an Autonomous Vehicle Management System that supports both fleet administrators and end users in interacting with a fleet of autonomous vehicles. The system should be designed to make recommendations about fleet composition and manage deployment, booking and payment functions. The system should accommodate the needs of users who are visually impaired or hearing impaired. In the first submission **(A1.1)** students were asked to prepare to name and describe the stakeholder groups and to provide a set of 10-15 appropriate questions for each stakeholder. In the second submission **(A1.2)** students were asked to interview the stakeholders and summarize answers to key questions. The instructor took on the role of the person asking for the system to be built. For each of the other type of stakeholders, members of the teams were instructed to take turns, with one member taking the role of the stakeholder and the others conducting the interview. The report from this phase included answers to the interview questions, explanation of risky features, a UML use case modeling

diagram, and use case descriptions. In the third submission **(A1.3)** students were asked to provide multiple use case input/output scenarios to show their understanding of the functionality of the system, for different stakeholders and their use cases. All materials were then submitted as a single professional requirements document for a grade. Students received detailed written feedback after each submission. A1 is not explicitly utilized as a data source for this study, but was a necessary guide to complete the other assignments and appeared during the interviews as a source of difficulty for the students.

Our evaluation of the approach of the student teams to design, the obstacles they encountered and the misconceptions they exhibited is based on three sources of data:

Assignment 2, part 1 (A2.1), the purpose of which was to produce a design of the system whose requirements they analyzed in assignment 1. The goal of A2.1 was to identify a set of application-level classes that would be needed to develop the system and to show the relationships among the classes using one or more class diagrams. They were also asked to provide short, high-level descriptions of the functionality of each class in their designs. Assignment 1 had also asked them to describe the system inputs and outputs for important use cases. This typically meant an input of a rider requesting a ride through the mobile app interface and an output of a car arriving to pick them up, deliver them to their desired address and a payment occurring. Teams were also asked to provide preliminary visuals of what a rider and fleet administrator might see, and provide a summary of how their system design supports the interactions in the use cases and their visuals (screen mockups). Finally, they were asked to provide five other UML diagrams (some combination of sequence, state or activity diagrams to clarify aspects of the design corresponding to key risk areas or features that seem hard to get right. Students submitted these documents for detailed feedback but did not receive a grade. However, A2.1 is useful to examine in that misconceptions and design difficulties may still manifest.

Assignment 2, part 2 (A2.2) asked the teams to refine and improve all their answers to part 1: to elaborate on the functionality of the classes in the class diagrams by describing public interfaces, including attributes and operations, to revise and improve their visuals, descriptions of inputs and outputs, and design verification (summary of how the system supports the use cases seen in the visuals and represented in the provided example inputs and outputs). They were asked also to provide descriptions of key operations in the form of informal requirements and guarantees.

The interview

After students submitted A2.2, we conducted exit interviews about their design decisions. Study protocol was approved as Exempt by the university's Institutional Review Board. Students read and signed consent forms so that the interviews could be recorded and the data could be used for research. They were informed that participation in the interviews was not mandatory, and were incentivized with a quiz grade for full completion of the interviews. The groups scheduled times to participate in the interviews and met with one of the researchers. The students were asked to do the following:

1. Explain their UML class diagrams.
2. Walk through a standard use case of their system.
3. Walk through a non-standard use case of their system (e.g. A visually or hearing-impaired person using the system or a car needing maintenance).
4. Describe some of the challenges faced in the project throughout the semester.

The groups' project documentation was provided on a computer and groups were informed that they could use the documentation to help with the explanations and answers to the questions. The group members took turns being the primary person providing the answers for the first three questions, with other group members encouraged to add anything they deemed relevant. The final question was presented as a free-for-all, with any and all members who had opinions encouraged to share them. The audio for the interviews was recorded and also the computer screens were recorded. Each interview took no more than 25 minutes, however, students were given as much time as they felt necessary to examine their documentation while answering the questions. The audio files were transcribed into interview scripts, with a different style of text; regular; underlined; and *italics* representing when a different group member began speaking on a particular question.

Following the transcriptions, the authors went through the videos together with print outs of the groups' documentation and the transcripts. We took notes on instances where the explanations given did not match the documentation, attempted initial categorization of common mistakes made in how students were thinking, and also began looking for interesting language within the transcripts. We also categorized direct statements for question 4, which probed at the challenges of this project-based, upper-level CS course.

6.4 Results

For the results of this work, we examine 8 groups of the 13 who were present in the course. These 8 were chosen because they were all full teams (three students) and all of the group members were able to attend the in-person interview. We present the findings qualitatively in a case study format, reporting our notes on the audio from the interviews with emphasis on the evidence of difficulties with design and specification and a brief discussion strengths of the submissions and interesting aspects of their difficulties.

6.4.1 Group 1

Difficulties with design and specification: This team experienced difficulties with several aspects of the design and specification task. For example, they had difficulty representing **abstract concepts** such as a *ride* to capture the temporary association between a car and a rider. They also failed to depict or describe the phenomenon of multiple simultaneous instances of riders, though an array of cars was indicated. Based on feedback they received on A2.1 they added class support for the notion of a ride, but still failed to represent non-physical phenomena such as startup, shutdown or a history of transactions.

The students described the difficulties they encountered with OO-design given the scale of the project as compared to their previous experiences with software design and specification: "In class we focus on employee relationships and an employee has a manager, and that's simple enough. Here we had several independent systems communicating with APIs as well as external databases ... and that was just the class diagram." Choosing an appropriate **level of detail** was a challenge, with one student stating " and at that point you don't know how much detail to go into for each one. ... It's just hard to distinguish the lines of where is ... like a good point."

We observed some attempts to **avoid complexity** by creating what we term a *megaclass* or *megaDB*, (a class or database with vague claims of performing many important functions or structuring and storing data). For example, the team described that the database has "special operations," hidden within its infrastructure such as the ability to optimize the deployment of vehicles to riders or to merely write that things were handled by "the system." Reflecting on this in the interview, team members stated, "...and then in requirements analysis we were focusing on actors. So the system just did everything."

We observed issues with **completeness** and with **consistency** between different elements of their specification. Although they described a UI interface, no class was presented that implemented that interface. In some of the documentation and in the interview transcript they mentioned the existence of a map interface to visualize where the car was and also the functionality to cancel a ride should a user decide not to need the car. However, neither of these elements were present or supported by the class diagrams or descriptions the group created. Interestingly, a member remarked that the group "spent a lot of time talking about the user interface and how we needed to show that on the UML," suggesting that they were aware of the need for consistency, but did not execute this when creating their final product.

Although the group indicated that they understood that they needed to represent the flow of control via a main method or event loop, they exhibited confusion between interfaces and classes, and stated that all of the UI logic was implemented in a Controller interface.

Discussion: This team was generally good at envisioning user interactions and provided reasonable mockups of user interfaces and good textual descriptions of use cases and user interaction. They were able to appropriately use activity and state diagrams to capture important features of system behavior. To the extent that their underlying system supported each use case they were able to document system behaviors associated with those actions, Their struggles with choosing appropriate levels of abstraction and managing the scale of the project led them to express a desire for more practice with medium-sized projects in the curriculum: "it almost felt like we skipped a step from the prior class to this one." They commented on the *impact of diagram creation* on their thinking about the design, saying "And also when you actually go to make the diagrams and figure it out, you realize you need so many more things." When asked about challenges faced, one member stated "I think it was too broad of a project," implying that the group did not have specifications scoped narrowly enough. This issue would be something to remedy while going through the requirements analysis phase, as that is the phase where the project is scoped out.

The class diagrams submitted by this team revealed some misconceptions about interfaces and how they are used. Their class diagram made use of three interfaces, a UI interface, an Account interface and a Management interface. One issue is that the Account and Management interfaces contained attributes, One explanation is that students were confused by the multiple meanings of the term interface (user interface versus programmatic interface). Another explanation is that their

lack of experience with implementing user interfaces left them "stumped" about how to represent it and so they merely left it out. Their sequence diagrams revealed confusion about active versus passive objects, or perhaps about their understanding of the meaning of activation bars in a sequence diagram: the admin role actor, an adminUI instance and an adminAccount instance were all shown with activation bars that ran the length of their timelines. However, none of the roles were annotated to indicate that they were active objects (contained their own thread of control) nor does it make sense from a design perspective that they would do so.

6.4.2 Group 2

Difficulties with design and specification: The class diagram produced by this group focused on representing physical and human entities but **failed to capture abstract concepts** such as a ride or collections of cars or users, although their diagram did specify multiplicities for each of these classes. Instead, they attempted to **avoid complexity** by defining a *megaclass* labeled System that would "facilitate communication between all other classes" and "have a data structure for the riders and admins" among other features. After feedback on part 1, the team added a main method to System, but the specification still lacked any attributes or other methods. In the interviews, the group described another approach they took to avoid addressing complexity in their specification of the system: assigning functionality to human actors, by "giving more power to the admin."

Their documentation was **incomplete**, lacking representations of UI-related classes, or meaningful representation of non-physical entities and phenomena such as the the flow of control, support for collections, and history of transactions. Their documentation **lacked consistency** across the multiple representations and verbal descriptions. They were aware that their representations and descriptions should be consistent, with one member stating "All I knew was that we needed to have functionality that corresponded to every function we had in our class diagram" when referencing the UI visuals. Even with this understanding, the group struggled with that consistency in the submitted documentation as well as the features discussed in the interview. The subject of payment was addressed in the interview but not supported in the UML class diagrams or descriptions, although it is depicted to some extent in activity and state diagrams and was represented in their UI visuals. Other UI visuals depicted History and Settings buttons, which were described in the interview, but again these features were not supported in the documents. Another example of inconsistency was a feature mentioned in the interview but not present in any design documentation:

the group mentioned that a "beep would come from the phone" like a "metal detector" as a way to alert visually impaired users that their car had arrived.

Discussion: This team was able to provide a detailed description of the UI and support for visually impaired users. Their class diagrams provide support for the identification of visually impaired and hearing impaired users via the specification of subclasses of the Rider class, and the features are described somewhat in their use cases. However, they provided no details of how they would support these features, as their submission lacked the description of any classes for UI support. When discussing the difficulties faced for the project, the team members referenced the impact **feedback** had on their work at multiple times, running into situations during design where they wondered "how are we gonna implement this part?" but then they "received feedback about something" and realized "Something's missing." Although the feedback was helpful, the students still struggled to move beyond classes tied to physical entities.

6.4.3 Group 3

Difficulties with design and specification: In A2.1, this group submitted a document with a UML class diagram that captured physical entities (users, cars, fleet owners) but failed to appropriately represent non-physical phenomena such as a ride, a user interface, flow of control, support for optimization, or the notion of collections of cars and riders. In an attempt to capture the notion of a ride they depicted a 1-to-1 relationship between cars and riders, which fails to account for the transient nature of the association or that over time a given rider may ride in many cars and a car will transport many riders. Based on feedback they received, they updated their design to include a GUI class, a Ride class and System class. However, the GUI class had no connection to underlying system information, and the Ride class had an association with riders and the System but not with cars.

To **avoid complexity** this group specified a *UseAlg()* method in the System class that purported to handle all of the optimization and functionality intended to automate the fleet of cars; i.e., the System class was a *megaclass*. In the interview, a team member stated, "...and the system would use the location of the user and run the algorithm to determine which car to send..According to each car's location and gas level and all of that..Which would also be stored in the car's database." When attempting to discuss the system further, there seemed confusion in the details, with a remark

of "So I guess in our system, there's a queue for user..It'll use the algorithm that's applied from uhhh.."

This group also had difficulty with **consistency** and **completeness**. For example, this group did not include the necessary variables or elements to store/input payment in their UML class diagram, but simply a makePayment() function. They discussed a "priority queue" and algorithm that determines which cars are sent first if more users request a car than are available, which is not represented in their UML class diagram. The group included a GUI class in the UML diagram, but it does not have access to the necessary system information to show the user when examining the connections of the classes. They also mentioned "I guess if you order, it would send a ping to the car" but the documentation contained no representation of how this notification or ping would actually work.

Discussion: Although this group created a set of syntactically correct UML diagrams, the content of the diagrams was overly simplistic, focusing on obvious behaviors. Their sequence diagrams revealed misconceptions about active versus passive objects and in interviews it appeared that they were unclear as to which objects should invoke which methods.

They struggled with scope, level of detail, and understanding how the different elements of the design needed to work together to support the application. One team member stated, "For me, one of the hardest things was trying to figure out what was communicating with what. ... Basically interconnecting the different classes."

They were uncomfortable with uncertainty in design and requirements and their incomplete requirements analysis led them to struggle with understanding the scope of the project, with one member saying it was hard "deciphering what was in the scope of us to do and what was outside of the scope." This confusion in the earlier parts explains some of the lack of consistency, as the project as a whole builds on previous parts. There is an implication here of the importance of feedback, as extensive feedback was provided between each assignment.

6.4.4 Group 4

Difficulties with design and specification: Group 4's A2.1 submission contained many of the same issues seen in other group's projects: only one rider and one vehicle active at any one time, a 1-to-1 rider to vehicle relationship, no support for the notion of a ride, a *megaDB* supposedly

capable of finding the nearest vehicle or deploying Emergency assistance, a failure to represent UI elements in the class diagrams, and no support for persistence across shutdown and startup. However, they responded to the feedback they received and by dint of conscientious effort and a solid requirements analysis in A1, produced a system specification that was consistent with the UI visuals and description and met the stated requirements. In addition, **good communication** among the team likely contributed to their success. During the interview, there were multiple references to communication, with one member who "would communicate to *name* or *name* and be like hey! Do you know if we have this in there and if not then let's add it." This good communication resulted in consistency that was not seen in many projects. This group even came up with a security feature of a QR code to scan to enter the car through interviewing stakeholders and communicating among themselves to think about the security problems that could exist for an autonomous car. Although not a required element, this practical and innovative design feature was consistent throughout their UI visuals, user stories, and UML class diagram.

Though this group managed to maintain consistency with their design, they still found the scope and level of detail a challenge, "getting more ideas as you implemented and realizing that Hey! I should probably need this function to make this work. And then just adding that on is just like a step by step thing that just like got complex." They also experienced difficulty in understanding the project as a whole, as "a challenge I had was just like conceptualizing. Like the whole system. Just the amount of overlapping everything." The group was not immune to the folly of avoiding complexity, discussing in the interview a car that "knows..Well the car knows kind of who's in the car and it adjusts things accordingly." This quote was in reference to interacting with a visually impaired user and suggests some complexity being handled just by the system that exists within the car instead of their own design.

Discussion: The strengths of this group centered around communication, feedback, and acting on that feedback, saying "the vehicle controller...where I think she mentioned something about the database couldn't do calculations." After receiving this feedback, the student "made that, and then realized I could use that with a bunch of other classes to get information." An important quote from the interview summarizing their design process was "And I was looking through the class diagrams and I was like Hey! Is this even feasible with what we actually have?" This ability to check for consistency despite the system's ever-growing complexity and ensuring communication between the

group allowed for many good design decisions and even interesting features not required for the assignment.

6.4.5 Group 5

Difficulties with design and specification: Group 5 was able to provide detailed class diagrams that lacked only a few of the important elements: persistence in the form of databases and startup and shutdown procedures, and a main method or event loop. To mitigate and **avoid complexity**, Group 3 spoke of an artificial intelligence (AI) that seemed to handle many complex elements. A member stated "The AI will automatically determine which vehicle is close to the customer and it'll dispatch that vehicle to that customer." This AI *megaclass* was not represented **consistently**. Although it was present in the maintenance scenario, it is not represented in the class diagrams or their descriptions. Another example of complexity avoidance occurred when a member discussed the idea of cancelling a ride, saying "he/she may decline it and we'll automatically send it back to the database." The database is presented as a *megaDB*, with functionality to analyze a record of rides and optimize the system, but its existence is not documented.

As described above, this group had issues with **completeness** and **consistency**. In the interview, a member said that the car "will disable itself from the queue of receiving requests" when a maintenance issue is occurring, but this functionality was not present in the UML class diagram. Once a maintenance issue was resolved, according to the group "everything is recorded in the database and the service is completed, the final report will be created and sent up, and then the vehicle will be back on the queue." This functionality also is not supported by the UML design and shows inconsistency between what the group believed they had designed and what they actually designed.

When asked what they found most challenging, this group found "the hardest part was trying to envision how we would realize the functionality that was being requested of us" and "capturing minor details. Cause it's like..It's so abstract." Translating the requirements to design details proved challenging as was conceptualizing a system that was more theoretical than concrete. The latter point was addressed further as a member said "to envision, you know with this not being an existing technology..How do we envision this being used? Because there's not really anything you can equate it to."

Discussion: Group 5 was able to document user cases with reasonable visuals and descriptions. They mentioned that they also had difficulty in the requirements analysis phase, mentioning that "it was hard for us to come up with the questions for the interviews." Similar to a few other groups, Group 5 could benefit from more experience with abstract concepts and more experience with the design of small to moderate sized projects to give them experience in evaluating the relative merits of various design choices and to gain additional hands-on experience with the properties of good design.

6.4.6 Group 6

Difficulties with design and specification: Group 6 was able to achieve a reasonable level of abstraction, after some struggle. As they describe, they "started with a lot of physical things. Like the customer and the fleet" when designing their UML class diagram and then "thought about more abstract things like the actual ride itself or the payment." They compromised on choosing a level of detail at which to represent the UI and the associated controller by including only a select subset of the UI elements in the UML class diagram and in the visuals. Their representations lacked somewhat in **consistency**. For example, their UI visuals and descriptions provided detailed scenarios of use by visually or hearing impaired users, and in the interview a group member said "on this screen, they will be given the accessibility needs that they need to be met." However, there was no notion of an impaired user or accessibility settings in the UML class diagram or class descriptions.

Similarly, they mentioned in their class descriptions that users would login to the system, and in their interview a group member stated "the user would log on to the app. So they have to create an account beforehand if they want to log on." However, no visuals or user scenario descriptions captured this aspect, nor was it represented in the UML class diagram

When asked about challenges, group 6 stated that they struggled with the amount of detail and abstract overall concept of the system. They felt "we have this really intricate and in depth system..But we really don't know how to create these algorithms and functions and such that would actually facilitate it." They also reported challenges in the earlier requirements analysis phase, not realizing that they could interview a variety of stakeholders, not just one per category, but then they realized and decided that "adding in additional stakeholders and not limiting it" made sense.

Discussion: Feedback proved very valuable to this group from the beginning, stating that "when

we got feedback on it we started to think a lot more about like actual point A to point B..How to actually design a system that would do something." Another member explicitly stated "I liked the fact that the first assignment every time was for feedback. That was a BIG big deal" in terms of knowing "what direction we're supposed to go is big for a project that's all-encompassing like this." Even with useful feedback, "it was kind of hard to understand what class uses what class when we don't really know how the classes work." Overall, this group performed well as they were able to successively refine their specification to an appropriate level of detail.

6.4.7 Group 7

Difficulties with design and specification: The A2.1 submission by this group had some issues: the need for container classes for the vehicles, a vehicle database, the ability to startup and shutdown, maintaining history and support for the UI elements. They were able to respond to this feedback and produce a detailed, high-quality design and specification document. This group handled complexity and managed to represent it within their design documentation, showing no signs of avoiding the necessary elements of design. They expressed that they felt that "there were always..fairly specific criteria that were being looked for in our assignments," although the group was not always sure of the scope until they received feedback.

Discussion: Group 7 for the most part was very detailed and consistent with their designs. They had only one instance of inconsistency, mentioning that when there was a major maintenance issue with a vehicle the server "sends a message to the car database." This functionality was not represented in either of the provided UML class diagrams. However, these diagrams were so detailed that the absence of this functionality was likely an oversight rather than an example of complexity avoidance. One potential criticism is that they may be guilty of overkill, providing so much detail that it is difficult to absorb the key elements of the design. This group had the server API act as "the brains of the system...making most of the decisions." They expressed some struggle with understanding the server concept originally, because "there's a lot of logic going between these different systems" and one member found it "very difficult having never had to do that before," but the group rose to the challenge and through feedback they were able to properly scope and design the system. This work included "a lot of research" on things such as the model view controller pattern and associated diagram. Overall, it seems as though the time and effort this group put into the project paid off, as

they managed to avoid many of the problems that other groups had.

6.4.8 Group 8

Difficulties with design and specification: The A2.1 submission by this group failed to provide sufficient detail. Their UML class diagram contained only thee classes (Administrator, Vehicle and Passenger) and two associations, connecting the vehicle class to the other two classes. The class descriptions were rudimentary, but they were able to provide reasonable scenarios of use, UI visuals and UML activity and sequence diagrams describing the scenarios of use. Other feedback warned them against the specification of a *megadb*, which they represented in the activity diagrams. They acted on some of this feedback, adding databases, additional classes and appropriate methods to their UML class diagram and expanding their class descriptions. However, their design still lacked a representation of flow of control, startup and shutdown, or a class that captured the notion of a ride (though they did have a database for ride history). They continued to struggle with finding an appropriate level of detail, with one member stating

One big example of lack of **consistency** is found between the UI visuals and the UML class diagram. The UI shown in the documentation and discussed in the interview is completely disconnected from the class diagram. User interface was looked at as a separate entity. Another example of inconsistency is found when looking at a theoretically nice feature of letting the system know when a rider has entered or exited the vehicle. According to the interview, "you'd press the button when everyone has exited the vehicle...which gives the vehicle permission to drive off." This feature could address security concerns, however it is not supported in the documentation. A similar disconnect was found between the passenger services and vehicle services classes.

Discussion: Group 8 mentioned many facets of how they felt they could have better succeeded at the project. The value of feedback arose, with the group feeling that they could not move forward to the next step without feedback on the prior submission.

These comments spoke to the need for feedback and how feedback earlier in the process would have helped them. Interestingly to note that "None of us went to her office hours" so there is an element of lack of communication with each other as well as the instructor over the assignment that could play a role in their concerns about feedback.

6.5 Conclusions and Limitations

This work revealed various insights into students' reasoning about design and what might cause some of the difficulties. A brief summary leads is that "design is hard," but we expand upon this idea below.

A common issue noticed throughout this study was difficulty figuring out the level of detail necessary for successfully designing the system. Many comments and observations seem to suggest that this difficulty relates to translating the subjective nature of this process (gathering requirements, figuring out what is necessary or useful features in the system) to the more objective side of designing class diagrams that are consistent with the features the groups decided to design. There is also an element of students' expectations differing from a standard CS course, where instructors typically provide students with the specifications or requirements of assignments. However, for this project, it is necessary for students to do the requirements analysis phase on their own, leaving room for them to not have asked questions needed to fully scope out the elements needed for the project. Students seemed to have an easier job with creating the use case scenarios and the UI visuals to represent those scenarios than with the design of the underlying system and capturing that design in a collection of UML diagrams and explanatory text. Even within the class diagrams, representing physical objects such as a car or rider proved less daunting than attempting to represent an abstract concept such as a ride or even the multiplicities necessary to connect the classes in the system. Abstraction difficulties in OO CS work were also found by Thommasson [204].

The issue of complexity avoidance persisted and was observed in many of the groups. A common strategy used to avoid complexity, the *megaclass,* relates to work on software design antipatterns such as the "God class" [168] and the "blob" [18]. Students created classes or databases with extended functionality to work around the complexity of the design problem. Complexity avoidance in CS is an observation also noted in Lawson's [119] work on students at various levels of undergraduate study attempting to solve a concurrency problem and Kennedy's [105] work in which students reverse engineer a program based on an executable while following a think aloud method [61, 141]. An explanation that lines up with previous related work would suggest fragile knowledge[158], which is a result that was also found in other work exploring student reasoning in CS topics [104, 105]. In evaluating the interview and transcripts, we identified instances where group members responded intuitively with features they believed they had designed or felt made sense, using the S1 part of

thinking as seen in [83]. Using this intuitive thought process when designing the diagrams for the system may have led students to unintentionally avoid complexity. Alternatively, students may have been using the deep thinking of S2 after realizing that there was not an intuitive way to create the overall system design, and used strategies of hiding or avoiding complexity to work around the lack of an apparent simple solution. If probed further as in a task-based interview[97, 47, 35], or given more time to examine their design during the interviews or in follow-up interviews, students might have used S2 and noticed that the created designs did not justify the features they were claiming during the interviews or use case scenarios.

We also observed issues with the completeness of the design and inconsistencies. Students were generally able to produce UI visuals that captured their user stories, but had many inconsistencies between the UI visuals, user stories, verbal descriptions from the interviews, and the UML diagrams they designed. Factors related to this issue include scope and level of detail, with groups not fully understanding the extent of the system to be designed nor the granularity at which to represent it. Kennedy [105] has also looked at design and how students scope problems while trying to code "in the wild." We believe that some of this issue relates to insufficient requirements analysis and failure to utilize available resources.

To follow up with this idea, another observed insight is the value that interacting with other people can bring to the success of design. Groups consistently mentioned the importance of receiving feedback through interactions with the instructor, but a benefit that some groups had and might not have realized is communicating and interacting with each other. This open communication allowed teams to stay more consistent with the various design elements and allowed the justification of features derived from requirements analysis to be better represented in the class diagrams. The importance of feedback in CS and programming has been examined in [120, 160] and Gusukuma [80] even did work to address improving compiler feedback to base it more on misconceptions. The ideas of communication and feedback being important echo software engineering in industry valuing keeping stakeholders involved throughout the development process to ensure there is not miscommunication on what is wanted compared to what is being developed.

Based on the observations of this study, suggestions for pedagogical improvements arise for teaching OO design. The first is to allow students to immerse themselves in the process of translating subjective requirements into more objective artifacts early on and on projects of increasing scale. Another is to emphasize the importance of interacting with people particularly in work that is team-

based. Instructors generally advertise their office hours, but there should be a push to ask questions to clarify any unclear feedback and to ensure that as a team, members are communicating regularly, even if it is necessary to be remote (video conference call or some other format).

This work was limited by not having access to the students after they finished the course to attempt to follow through with more questions. The researchers also acknowledge the context of this work, occurring at a large, public US institution, which affects generalizability. Students in other geographical locations might have more exposure or experience to team environments in early education or within the early years of CS higher education that could mitigate some of these observed issues. It is our hope that this work can be used to inform pedagogy for instructors that see its merit and offer researchers a stepping stone to build upon in expanding the knowledge and understanding of student reasoning about CS topics.

Taking into consideration the issues with requirements and resulting scoping difficulties, a suggestion would be to provide more practice with short turnaround on micro-versions of requirements analysis and associated design and specification before engaging students in a larger project.

Chapter 7

Misconception-Based Peer Feedback Intervention

The results of the formative studies suggest a number of possible interventions. After observing the role that feedback played in all of the formative studies, reflecting on the related work surrounding active learning, conceptual engagement, and ways to incorporate feedback into an active learning technique, I decided that a promising intervention would allow students to have structured feedback that offered discourse to activate the problem solving elements of programming, allowing the students to "make explicit that which is implicit."[186]. This technique would offer a way to address the scaling of CS courses by being low resource-intensive, and would ideally help address retention rates if it were to have widespread adoption. This intervention is an active learning technique that I have labeled misconception-based feedback (MBF). The basic format is that one peer (**A**) is provided prompts to ask another peer (**B**) about code that **B** has written. **A** has attempted the same programming assignment and is thus familiar with both the problem and with the implementation approach, associated challenges, and design decisions. The coding problem employed in the intervention has been designed to exercise the concepts related to these misconceptions. The prompts are structured to give **A** and **B** an environment conducive to productive conversation centered around misconceptions that have been shown to cause difficulties with programming. While discussing the prompts, the pair can also work to improve the code. This technique is supported by Chi's self explanation study[30], the benefits surrounding pair programming[214], and the rubber

ducky debugging methodology[32]. The MBF technique is made using a "recipe" that incorporates the following ingredients:

1. Identify Misconceptions
2. Design a programming assignment to exercise the concepts associated with those misconceptions
3. Develop structured prompts to address those misconceptions

The following subsections walks through these steps in general, framing them with an example before describing the evaluative study that I used for the purposes of this dissertation.

7.1 Identify Misconceptions

The first step in the MBF technique focuses on the "M," or misconceptions. To design a technique that addresses misconceptions, you must first know the misconceptions to address. This can be done a variety of ways. One way could be through literature reviews. A growing body of work in the computing education research (CER) community addresses misconceptions related to concepts in computing. An advantage to using a literature review approach is that it can save time by using already well-defined misconceptions that apply to your topic/s of interest. A disadvantage to this approach is that you cannot ensure that the misconceptions apply to your population of students without performing some testing.

To address this problem, another way to identify misconceptions is conducting studies meant to identify misconceptions about a topic or set of topics. Available tools such as CS concept inventories[201, 148], or modifications of them such as my "EYR" study[104], and observing how students reason while they program as in my "coding in the wild" study[105] are helpful in identifying misconceptions and the causes behind them. For example, I focused on introductory programming misconceptions and narrowed that scope through related work on identifying important and difficult CS topics[78] and cross-referencing that work with the context and scope of the curricula for the courses that were a part of the studies (CS1 + CS2) and topics found in the ACM curriculum guidelines[99]. After developing my "EYR" assessments, I administered them and used the data to determine misconceptions relating to *Pass by value*, *Scope*, and *Same name, same location*, as seen in Table 7.1. My "coding in the wild" study then observed how students reasoned

while attempting to implement a program, and from that, the other misconceptions in Table 7.1 arose. It is important to note that the misconceptions identified need not to be an exhaustive list, as the process of identifying and understanding the reason behind them can be time consuming. You should define a scope of misconceptions you are interested in exploring and focus on misconceptions either supported by research or indicated by local context. A way to quickly gain insight into the concepts your students are struggling with is analyzing online exams. Tools such as Canvas provide breakdowns of the questions that students answered incorrectly the most often and also the most common incorrect answer choices chosen (assuming the questions are multiple choice). For the MBF technique, the defined misconceptions are the metric used to determine the learning outcomes your students exhibit. These misconceptions become codes to be applied to an EYR pre-test and post-test (explained in 7.4) through closed coding. After you have identified the misconceptions you wish to address, the next ingredient to prepare is a programming assignment to address these misconceptions.

7.2 Programming assignment to address misconceptions

Once you have obtained the set of misconceptions for which you qualitatively code, you next need to design a programming assignment that allows students to attempt to use the concepts associated with these misconceptions. This part of the recipe allows students to practice the concepts and also provides them with a (hopefully) working artifact to discuss once it is time to participate in the MBF technique. It is possible to provide students with pre-made example code that contains the misconceptions and use the prompts to have students analyze the code for this step, although it would detract from the students constructing their own knowledge and being able to work using their own misconceptions as a guide. To ensure that the assignment thoroughly addresses the concepts related to the misconceptions of interest, you should complete the assignment yourself to see where students might use them. It is also wise to pilot test with other students or less experienced programmers to determine approximately how long it takes to complete the assignment. The programming assignment can be given as a lab exercise, classwork, homework, or project to work on. The assignment should be completed individually if possible. An example programming assignment could be the short calculator program used in my "coding in the wild" study[105]. This program requires students to create multiple functions, with the ideal solution being implemented

Table 7.1: Misconceptions

Misconception:	**The student thinks...**
Pass by value (PBV)	operations on the copy affect the original
Void function - no "return"	void functions return a value despite the lack of a return statement
Void function - "return"	void functions with an empty return statement return a value
Scope	local variables can be accessed outside the scope of their declared function
Same name, same location	variables in different scopes with the same name are the same variable
Pass by reference (PBR)	passing a variable by reference cannot result in a change to the original variable
Arrays not PBR	the passing of an array does not follow PBR semantics
Void function not PBR (Arrays)	void functions with passed arrays do not follow PBR semantics
Pointers not PBR	pointers do not follow PBR semantics
Void function not PBR (Pointers)	void functions with pointers do not follow PBR semantics
Pointers	pointers are references or vice versa

using PBR for at least one of the functions. The functions take in two integers and either add them, subtract them, or flip the signs of them (change positive to negative and vice versa). These functions provide students the opportunity to work with the PBV and PBR-related misconceptions defined in Table 7.1. The program also tasks students to have the code run in a loop and count the total number of operations performed, which gives them the opportunity to work on the scope concepts. The final ingredient of the recipe adds in the "secret sauce" in the form of structured prompts to address the misconceptions.

7.3 Structured prompts

To explain the construction and use of misconception-based feedback prompts, I describe the prompts that were used in my associated study, which is described at the beginning of Section

Figure 7.1: MBF Index Cards

7.4. The group that used MBF receives feedback from a peer, in the form of prompts based on misconceptions identified in my formative studies (Chapter's 3-6) that allow students to discuss common issues that might arise and explain the decisions they make during the simple calculator task with a partner. The questions are broken up into categories:

1. Overall Program
2. Pointers
3. Arrays
4. Operation Count
5. Code Cleanup

The index cards are put together in a way so that each category was a section, and partner can go to different sections depending on how **B** decides to implement the solution to the flip signs program. The full list of prompts can be seen in Appendix G. Regardless of choice of implementation, the expectation is to eventually go through the whole set of cards, with discussions based on how a person can theoretically implement the solution using arrays or pointers if **B** does not use that option. The questions are designed to prompt conversation between the partners about potential misconceptions. **A** is instructed to ensure that the dialogue was conversation, with **A** being able to communicate his opinions and not just ask questions that **B** is meant to respond to. **B** is instructed to make changes if she feels they are appropriate based on the discussions. The following subsections walks through the arrangement of the sections of index cards, providing a few examples of the prompts that are in each section.

7.3.1 Overall Program Cards:

The prompts begin with some basic questions to ensure students understand the control flow of their own program such as "Can you walk through the code/function?" Then there are questions to check the syntax of the overall program such as "Where are variables declared?" and "What are their types?" From this point, questions become more conceptual and focus on the required function, asking "How do you send values (pass parameters) to the flip function?" and "How are the flipped values accessed from the main function?" In between conceptual questions, there are still more questions to check syntax for declaring the function, its return type, etc. The final prompt states to "Reflect on your solution - did you use pass-by-value? Pass-by-reference? Something else?" The lack of understanding of pass-by-value (PBV) semantics is a very common misconception seen in my EYR study[104] (Chapter 4) and misconceptions relating to PBV and pass-by-reference (PBR) were found in my "coding in the wild" study[105] (Chapter 5). This prompt causes much of the fruitful discussion that seems to result in improved learning outcomes, causing multiple of the MBF pairs to reflect and attempt to explain what pass-by-value and pass-by-reference are and how they work. When the pair reaches the end of the Overall Program cards, they are directed to either the Pointers or Arrays cards depending on how they attempt to implement the solution. If **B** uses neither arrays nor pointers, **A** is instructed to go through both sections of cards and **"discuss how the solution might be implemented using the questions as a guide."**

7.3.2 Pointers Cards:

This section of cards follows a pattern similar to the Overall Program, starting with questions like "Where are the pointers created?" and "What are they pointing to?" These questions ensure that students know the basics of pointers and that they point to an address. The questions then get more conceptual, asking "Do you pass an address or a value as a parameter in the function call? What's the difference?" and "How are the addresses and their values accessed and/or changed inside the function?" These questions require the students to think explicitly about the two elements of a pointer variable (the address and the value) and discuss how to manipulate the values. When the pair reaches the end of the Pointers cards, **A** is instructed to go through Arrays if time is available, and if the pair has already gone through Arrays and Pointers, move on to the Operation Count and Code Cleanup Cards.

7.3.3 Arrays Cards:

This section of cards also begins with ensuring the basics of declaration are understood, prompting "Where is the array declared?" and "What values are stored in the array?" After framing how the arrays are set up, more conceptual questions such as "Do you pass the array as a parameter? If so, how?" and "Are the values in the array different after the function call returns? How does that happen?" Similar to the Pointers card, the final prompt instructs the pair to either go to Pointers or if they have finished both, move on to Operation Count and Code Cleanup cards.

7.3.4 Operation Count:

This section of prompts ensures that the students properly implement the functionality of counting the number of operations performed and displaying it. It first asks "How do you know when to stop running the program?" It then asks "How do you keep track of how many times you performed the flip operation?" and finally asks "How and when do you display the count of how many times you ran the flip operation?" In many cases, these questions made students realize that they needed a counter variable because the counting operation is not explicitly in the instructions but rather in the functionality of the executable provided. The questions in this category do not elicit much conceptual discussion for this study.

7.3.5 Code cleanup:

The final section of cards focuses on avoiding wasted declarations, revisiting declarations, and considering duplicate variable names. The first question asks "Do you use all of the variables you've declared?" The next prompt focuses on scope misconceptions that students have, asking "Do you ever declare two variables of the same name in different functions?" The prompts follow up by asking how assigning a value to one variable affects another variable of the same name but in a different function. One of the prevalent misconceptions found in my formative studies was around the idea of scope. Students struggle with variables that are declared in different functions with different scopes and viewed them to be the same variable. This misconception is seen in both the EYR study and the "coding in the wild" study (Chapter's 4 and 5). The section finishes by asking if global variables are used and if so, where are they initialized, where are they used, and how do their values change, another misconception from formative studies.

The structured discussion around misconceptions as generally explained in subsections 7.3.1-7.3.5 supports the MBF technique. These prompts are developed on the basis of plan-like approaches to problem solving as seen in [211]. To develop these prompts, I look at the work of Bloom's taxonomy[1], the SOLO taxonomy[9], McCracken's problem solving framework for first year students learning to program[132], and Soloway's work on learning to program by learning to construct mechanisms and explain those mechanisms[186], all described in more detail in the following subsections. The idea behind the development of these prompts is to nudge students to make explicit the planning elements of programming that are implicit while trying to solve problems. In the following subsections, I discuss the related work behind the prompt development and how it led to the choosing of the prompts.

7.3.6 Bloom's Taxonomy

The content and order of the prompts in the MBF intervention are based in part on Bloom's Taxonomy. Bloom's Taxonomy is a tool to classify thinking based on its level of cognitive complexity[68]. The categories are arranged hierarchically, with each higher level suggesting a mastery of the levels below it. The six levels in the original version of the taxonomy are, from least complex to most complex: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation[1]. The revised version of the taxonomy changed the categories from nouns to verbs

and called them from least complex to most complex: Remembering, Understanding, Applying, Analyzing, Evaluating, Creating[1]. The definitions of the revised terms are as follows:

- **Remembering:** Retrieving, recognizing, and recalling relevant knowledge from long-term memory.
- **Understanding:** Constructing meaning from oral, written, and graphic messages through interpreting, exemplifying, classifying, summarizing, inferring, comparing, and explaining.
- **Applying:** Carrying out or using procedure through executing, or implementing.
- **Analyzing:** Breaking material into constituent parts, determining how the parts relate to one another and to an overall structure or purpose through differentiating, organizing, and attributing.
- **Evaluating:** Making judgments based on criteria and standards through checking and critiquing.
- **Creating:** Putting elements together to form a coherent or functional whole; reorganizing elements into a new pattern or structure through generating, planning, or producing.

This taxonomy helps to guide the development of the prompts. Within each of the sections of the cards, the general trend requires students to move up with respect to the level of Bloom's Taxonomy that characterizes their thinking. Take for example the Pointers Cards, which is where the MBF technique shows the most evidence quantitatively and qualitatively of improvement for the MBF group and can be seen in Figure 7.2. The first question prompts "Where are the pointers created?" This falls on the Understanding level of Bloom's Taxonomy, as the students merely need to explain where a construct (pointer) is created or infer where they would be created if the student did not use pointers to implement a solution. The next two questions move up to the Applying level, and asked "What are they pointing to? How did the pointer take on that value?" The students have to be able to implement pointers and not just understand the construct to explain what they are pointing to and how they take on values. A few questions later, students are asked " Do you pass an address or a value as a parameter in the function call? What's the difference?" This moves their thinking up to the Evaluating level, prompting them to judge the type of parameter passing they have implemented and potentially critique decisions made. The questions in this section

follow this pattern, with none of the questions placing the students in the lowest level of thinking (Remembering), but generally asking about their understanding and explanations of concepts first, and next ensuring they can apply that understanding, eventually prompting them to evaluate their understanding of the material. This pattern relates to Soloway's[186] work on students needing to learn to construct mechanisms and explanations. Soloway states that as scientists we must "make explicit that which was implicit," particularly when attempting to teach novices. Novices need the opportunity to explicitly state their plans and goals whereas experts already have plans implicitly in their minds as schema. This prompt development also ties into the Perkins study I described in Chapter 2[158]. Perkins categorizes fragile knowledge into four types: missing, inert, misplaced, and conglomerate[158], described in more detail in Chapter 2, using a controlled experiment. The results of Perkins' study found that better cognitive skills needed to be taught, focusing on problem-solving strategies to improve programming performances and address the fragile knowledge students exhibited. Bloom's Taxonomy also has a knowledge dimension that categorizes knowledge as factual, conceptual, procedural, and metacognitive[114]. Factual knowledge is considered the basic elements students must be familiar with in a given field to solve a problem. Conceptual knowledge consists of the interrelationships between the factual knowledge elements within a larger structure that allow them to function together. Procedural knowledge deals with the skills, techniques, and methods associated with how to do something. Metacognitive knowledge relates to the knowledge about cognition and awareness of one's own cognition. Using these structured prompts I have developed can help to expose missing knowledge and activate inert knowledge as defined by Perkins[158], allowing students to have a better grasp of concepts. I also have designed the prompts so that they explicitly have students explain their factual knowledge and procedural knowledge, combining these with questions that allow discussion of conceptual knowledge to occur and reflective questions that enable students to use metacognitive knowledge. Next, I discuss the related work surrounding the desired level of discourse for students participating in the MBF technique.

7.3.7 SOLO Taxonomy

For each question developed in the prompts, I considered the language it would elicit, using the SOLO taxonomy[9]. SOLO builds from Bloom's Taxonomy[114], which looks at educational objectives. The SOLO taxonomy is intended to categorize learning outcomes based on language used. The categories are as follows:

Pointers Cards:	McCracken	SOLO	Bloom's
1. Where are the pointers created?	Subsolutions	US	Understand
2. What are they pointing to? How did the pointer take on that value?	Subsolutions	MS	Apply
3. Where are pointers dereferenced?	Subsolutions	US	Understand
4. Do you pass an address or a value as a parameter in the function call? What's the difference? What is the type of the parameter/s passed into your function?	Abstract problem	Relational	Evaluate
5. How are the addresses and their values accessed and/or changed inside the function?	Abstract problem	MS	Analyze
6. Are any values in the main function modified as a result of the flip function's execution? How does that happen?	Evaluate	MS	Evaluate
7. How are pointers and references related? Different?	Abstract problem	MS	Analyze
8. Do you create any reference variables?	Subsolutions	US	Understand
9. Where are the reference variables created?	Subsolutions	US	Understand
10. Where are they assigned a value? What do they refer to?	Subsolutions	US	Understand
11. Do you use reference variables in passing parameters to a function?	Subsolutions	MS	Apply
12. How are the reference variables used inside the flip function?	Subsolutions	MS	Apply
13. Are any values returned from the function by using a reference variable?	Evaluate	MS	Evaluate

Figure 7.2: Pointers Cards

- **Prestructural:** The student manifests a significant misconception, or uses a concept irrelevant to programming.
- **Unistructural:** The student manifests a correct grasp of a part of the problem. For example, a student describes the functioning of one or two lines of code. (Describes the concept with a base understanding).
- **Multistructural:** The student manifests an understanding of most lines of code, but does not manifest an awareness of how the code functions as a single coherent whole - the student "fails to see the forest for the trees." For example, a student might translate each line of code into pseudo code. (Manifests an understanding of most of the concept, but not all of the details of usage are there - ex. gets that pointers can be used to pass by reference and change values, but is unsure how to implement or when/where &s or *s should be applied).
- **Relational:** The student manifests an understanding of the code as a single coherent whole, by describing the function performed by the code - the student "sees the forest." (Manifests an almost full understanding of the concept and can relate the concept to the programming task or use it in other examples).

The parenthetical parts of the definitions have been added to capture the type of discourse

that the MBF prompts are meant to elicit. These prompts were developed to elicit the misconceptions that students currently hold, but use appropriate terminology to define the concepts and move the discussion students have away from prestructural towards one of the higher levels such as unistructural or multistructural. Similar to the how the Bloom's Taxonomy level of cognitive complexity was considered when designing the prompts, the MBF prompts are also designed considering the type of expected discourse they would elicit. Using the Pointers cards as an example again, the first prompt, "Where are the pointers created?" is categorized as unistructural. This requires a cursory understanding of the concept of pointers to explain where in the code they are created. The next two questions, "What are they pointing to?" and "How did the pointer take on that value?" require a multistructural understanding of pointers. The students must know how the syntax is used and how the address and value elements of a pointer interact to allow pointers to take on a value. Several questions later, there is a Relational category when prompted "Do you pass an address or a value as a parameter in the function call? What's the difference?" These questions require the students to think about the whole picture of the use of pointers with respect to PBR and are intended to engage the students with a rich discussion of PBR vs. PBV, one of the common misconceptions students struggle with when learning to program. Just as the questions increase in complexity on Bloom's Taxonomy, they also go up the hierarchy of the SOLO taxonomy, with more discussion being expected/prodded for as the students are showing understanding of the basic concepts.

7.3.8 McCracken's Framework

McCracken developed a framework that can be used when determining learning objectives for first year students learning to program. This framework is as follows[132]:

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate

These stages are described in Chapter 2 and were used in my "coding in the wild" study (Chapter 5) as codes to determine the part of the problem-solving phase students were currently working on. For the prompt development, the questions were categorized by where in the framework they existed. Again, looking at the Pointers cards, the first question, "Where are the pointers created?" requires students to transform sub-problems into sub-solutions. Students must understand what pointers are and the syntax it takes to create them. The following questions remain in the sub-solution space until the questions "Do you pass an address or a value as a parameter in the function call? What's the difference?" These two questions are categorized as *evaluate* in the McCracken framework, as it requires the students to understand and be able to tease apart the differences between PBR and PBV. The next question has the students *abstract the problem*, prompting "How are the addresses and their values accessed and/or changed inside the function?" Students must take this language and be able to explain in terms of concepts and programming constructs how pointers are changing the values in a function. After showing an understanding of the problem, the prompt moves to evaluating the problem solving, asking "Are any values in the main function modified as a result of the flip function's execution? How does that happen?" This structure of questions ensures that students are first understanding the concepts and pieces of the problem and the sub-solutions necessary to solve it and then follows those types of questions with questions that evaluate their own problem solving abilities and steps. This relates back to McCracken's work on problem solving[132].

Building on the results from the McCracken study which suggested that students needed to focus more on understanding and breaking down the problem, these prompts focus on addressing these issues. The general flow of the prompts in each section goes from lower levels of cognitive complexity to increasingly more difficult, encouraging novice students to make explicit their implicit plans and problem solving strategies. The earlier questions in the section are designed to ensure that the syntax and basic structure of a given subproblem are understood. These questions check that students have properly declared the necessary pieces of the part of the problem they are currently working on. These prompts are generally in the Understanding level of Bloom's Taxonomy, the Unistructural level of the SOLO Taxonomy, and the *abstract the problem* or *sub-solution* levels of McCracken's framework. The prompts then ask about the details of constructs and elements necessary to solve the problem (declared variables, declared functions, etc.). These questions focus more on the subproblems and make students explicitly consider common errors such as mismatched types in their programs. This can address both the passing of variables into functions and those

variables being received and returned from the functions. These questions typically fall in the Applying or Analyzing in Bloom's Taxonomy, the Multistructural category of the SOLO Taxonomy, and the *sub-solutions* or *sub-problems* levels of McCracken's framework. Then there are questions that focus on making students plan out how they use the proper programming concepts. For this particular programming example, this generally means explicitly discussing how and when variables are passed into a function and how and when their values are changed/returned, depending on implementation choice. These prompts are in the Analyzing and Evaluating levels of Bloom's Taxonomy, Multistructural and Relational levels of the SOLO Taxonomy, and the Evaluate and Abstracting the Problem levels of McCracken's framework.

This general guide in sections 7.1-7.3 can be followed to design the MBF technique for other programming problems/projects. One method of implementing this process is in conjunction with a scheduled lab for a programming course. The programming assignment designed to allow students to work on the concepts of interest can be assigned the first week of a lab and then the following week the students can break into pairs and complete the MBF technique. Another use case is assigning the programming assignment in class and having students find a partner to complete the MBF technique with as a homework assignment (or vice versa depending on how much time the programming assignment would take to complete). Although index cards were used for how I administered the MBF prompts, it is possible to have the prompts printed and sectioned out on a sheet of paper for less overhead. The next section discusses how I have executed this MBF technique in my experimental design for this dissertation.

7.4 Research Design

This study leverages a key theme of feedback that was a theme in the results of the formative studies in Chapters 4-6 and attempts to compare the benefits of feedback provided by an autograder with feedback provided by peers. In Chapter 4, the task-based interviews that followed my EYR assessment survey found that students providing self-feedback through engaging in self explanation were able to address and remedy misconceptions when attempting to walk through code[104]. In Chapter 5, feedback's role was in students interacting with the compiler to successfully implement a program solution and also providing self-feedback through the think aloud protocol[105]. In Chapter 6, feedback was a core part of the design of the research, as students were provided feedback between

their assignment submissions and interviewed about the feedback. For this study, the misconception-based feedback (MBF) group is the treatment group and the human autograder (HAG) group is the control. For each group, students are broken up into **partners**, with Partner A (**A**) being the person who provides feedback and Partner B (**B**) being the person who receives feedback and also the person who has up their code. These **partners** are peers enrolled in the same course and taking the same lab as each other. The research questions that I address in this study are:

RQ1: Can a structured debriefing process based on misconceptions (misconception based feedback) improve student learning outcomes related to a programming assignment?

Hypothesis: MBF will show improved learning outcomes related to a programming assignment in the form of a reduction of misconceptions from pre-test to post-test.

RQ2: What misconceptions can be reduced using misconception based peer feedback?

Hypothesis: Misconceptions related to PBV and PBR, Scope, and difficulties with Pointers and Arrays will be reduced based on MBF.

RQ3: How do the roles of the students in the MBF structured process (feedback provider/receiver) affect their learning outcomes?

Hypothesis: The partners (**A** and **B**) will have similar improved learning outcomes as feedback provider or receiver since the MBF technique is structured for discourse allowing both participants to gain feedback, but **B** (feedback receiver) will have additional benefits because of their intellectual ownership of the code being used for the MBF prompts.

To test these research questions, the MBF study used students from the second semester programming course (CS 2) at an engineering-focused public institution in the US. Institutional Review Board approval was obtained and the study took place during the first week of classes in the Fall semester of 2019. The study utilized two 50-minute lab sessions, with students completing an "Explain your reasoning" (EYR)[104] pre-test at the beginning of the first session with questions focused on concepts of interest (parameter passing, return values, pass-by-value vs. pass-by-reference) and post-test at the end of the second session with similar questions but variable and function names and values changed. Students were given 20 minutes for the pre-test and 20 minutes for the post-test.

As seen in Figure 7.3, the other 30 minutes of the lab sections consisted of students working on a programming activity to flip the signs of two integers as described below on day 1 and either a the MBF intervention or HAG protocol using the code from that programming activity on day 2. For the first day, students received no assistance from peers or the instructors, unless that assistance

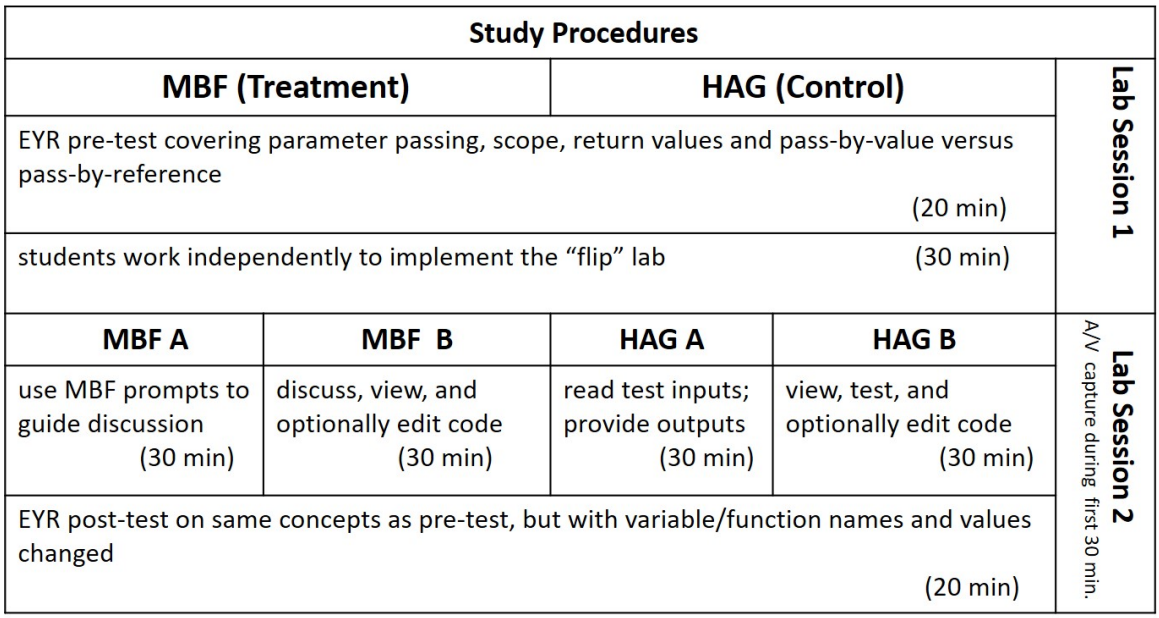

Figure 7.3: Study Procedures Overview

dealt with manipulating the programming environment so that the students could continue to move forward on the assignment. Students could leave when they successfully completed the task or the 50-minute lab session ended.

For the purposes of this dissertation study, the programming assignment (Section 7.2) students were tasked to complete was an abridged version of the calculator task from my "coding in the wild" formative study[105]. The students had to design a program that took in two integers and a separate function to flip the signs of those integers (change negative to positive and positive to negative). The program was to execute in a loop until a user entered an exit condition (an integer other than 1). After the exit condition was entered, the program should print out the total number of calculations performed. This programming assignment was chosen as it was shown in my "coding in the wild" study (Chapter 5) to give students the opportunity to work on the misconceptions found in my EYR study (Chapter 4). These misconceptions, seen in Table 7.1, were used to perform closed coding on "Explain Your Reasoning (EYR)" pre-tests and post-tests, as described in Chapter 4. The bolded entries in the chart represent the misconceptions that were present in the statistically significant quantitative results, as seen in Chapter 8.

I select an autograder as a point of comparison because it is a method that computer science departments and courses are adopting to address rising enrollments, and although it may save

time and resources[213, 57], it may not be providing benefits to student learning outcome[213, 57]. To control for the instruction style, the autograder is simulated using one of the partners (**A**) as the autograder. This decision minimizes confounding variables, as using an actual autograder would bring an issue of having two types of feedback (misconception-based and confirmation/correct-incorrect) and two types of instruction delivery (human for MBF and computer for autograder). These partners are peers who are enrolled in the same course and taking the same lab as each other. In this study, the MBF group is the treatment group and the human autograder (HAG) group is the control. This structured misconception-based feedback (MBF) that I compare against the autograder is an active learning technique I developed to guide students to directly address misconceptions that occur when programming and is described in 7-7.3.

The 2nd day of the study had the sections split into one of two conditions: human autograder (HAG) peer feedback (control group) and misconception-based peer feedback (treatment group). The HAG group had one partner simulate an autograder, providing only confirmation feedback and the other partner testing their code from day 1. This group is explained in more detail in the next subsection. The MBF group is as described in Sections 7.1-7.3 at the beginning of this chapter. For both groups, audio and screen capture recordings of the first 30 minutes of day 2 were gathered. For the remaining 20 minutes of day 2, all groups took an EYR post-test with similar questions to the pre-test. Students took these tests individually and were instructed to write on the post-test whether they were Partner A or Partner B.

I recruited the students in person, and the study took place at the same location and time as the students' regular lab sessions. The professor teaching the course agreed to count completion of the study as a normal lab grade for the course. The course comprised two lecture sections and four lab sections, with one of the lab sections designated for Bridge transfer students who did not have the same background experiences as the other three sections. This Bridge transfer lab comprised of students who transferred into the university from either a two-year college or another four-year institution and had completed an acceptable CS1 pre-requisite course but not the one offered at the institution used in this study. Considering CS introductory pedagogy does not have a consensus on first programming language to teach students, the students in this lab were mostly not familiar with the C programming language, the first language taught at the university and the one the programming assignment was designed using. Thus, the Bridge group data was not used in this study. Each lab section had approximately 30 students. Since in the human autograder condition,

half of the students would be providing feedback as the human autograders, only 15 students per section would receive feedback that could potentially help them learn. For the misconception-based peer feedback, the co-constructivism theory of learning would suggest that both the provider and receiver of this form of feedback would have learning benefits, but there is hypotheisized value from **B** being the partner who has his code in front of him while participating in the intervention. Therefore, the autograder group needed twice the number of sections/participants to have an even balance. This process was randomized by placing the three non-Bridge section numbers in a hat and selecting one to be the MBF condition. As the study had two conditions (human autograder and peer misconception-based feedback), I decided to use the Bridge section as a misconception-based feedback group. Their data was collected so that all students had the opportunity to be part of the study, but was not used for analysis as the population is very different. With respect to the populations of the other three sections, as they all needed to take the prerequisite course of CS1, the expectation is that the populations are similar. However, as every individual was not randomly assigned a condition but instead the lab sections were randomly assigned, it cannot be assumed that the individual students were equivalent. Below, I walk through the HAG group's experimental protocol.

7.4.1 Human Autograder (HAG) Feedback Protocol

The human autograder (HAG) pairs were split into Partner A and Partner B. **A** took on the role of the simulated autograder, tasked to only provide confirmation feedback for the test cases they were given on index cards. The front of the index card had the inputs that **B** should use and the back had what the output should be based on those inputs. Figure 7.4 shows an example of the front of an index card providing the proposed input. Figure 7.5 shows an example of the back of an index card providing the output for the input in 7.4. These index cards were designed based on running test cases against an actual autograder, so that the outputs provided were identical to what an individual would see if testing the given inputs. **A** was given a stack of 30 input cases, varying from running the program with just 1 calculation up to running the program three times using the required loop. The students were tasked to go through the 30 test cases provided to **A**, and if **B** did not have working and testable code, there were instructions to either switch roles or for **B** to edit and simplify the code until it was at a point where it could be executed and tested. The cases consisted of various options that would need to be considered such as negative numbers, numbers

too large for the memory of integers, zeros and negative zeros, and other cases. **B** was instructed to make corrections to their program as necessary based on the HAG feedback from **A**. Following the intervention, the pair was instructed to save and submit the video file that included the screen recording as well as the audio.

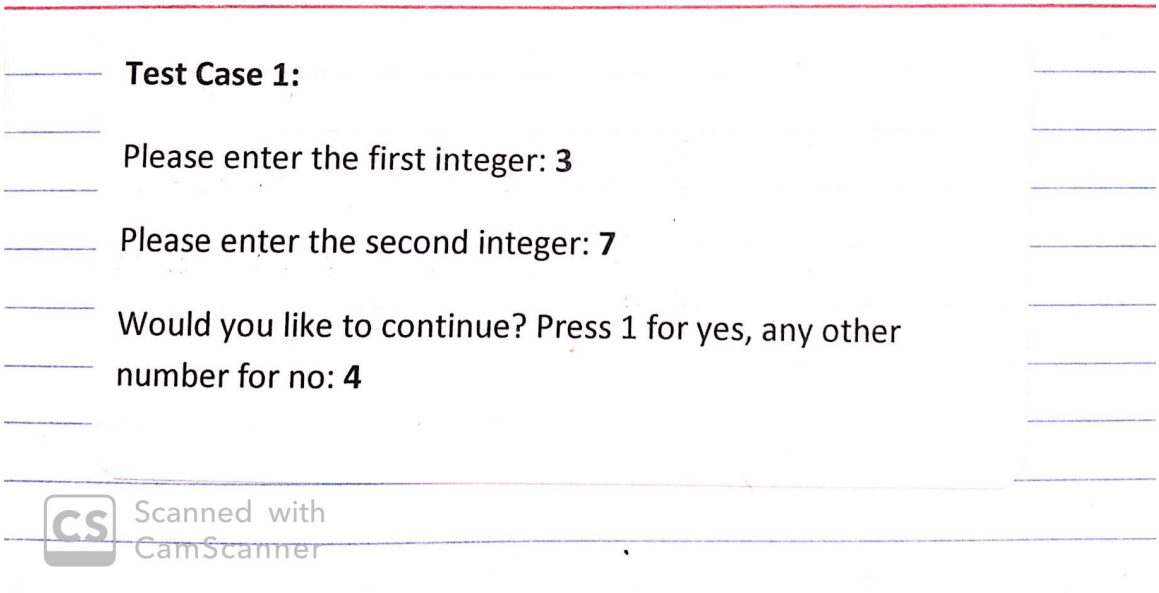

Figure 7.4: HAG Input

This group is meant to serve as a baseline and compare the misconception-based feedback intervention to what autograders generally provide for programming assignments, which is confirmation feedback. **A** was tasked to only provide feedback on whether or not **B**'s code worked properly, but the interactions were not monitored. Instead, audio was recorded of the interactions between **A** and **B** and analyzed to see if there were conversations that happened outside of the expected autograder dynamic. Although tasked with providing only confirmation feedback, it was possible that the human element of the HAG group would elicit conversation and some of that conversation could affect how students performed on the post-test. The next chapter discusses the analysis methods, results, conclusions, and future work suggested by the MBF study.

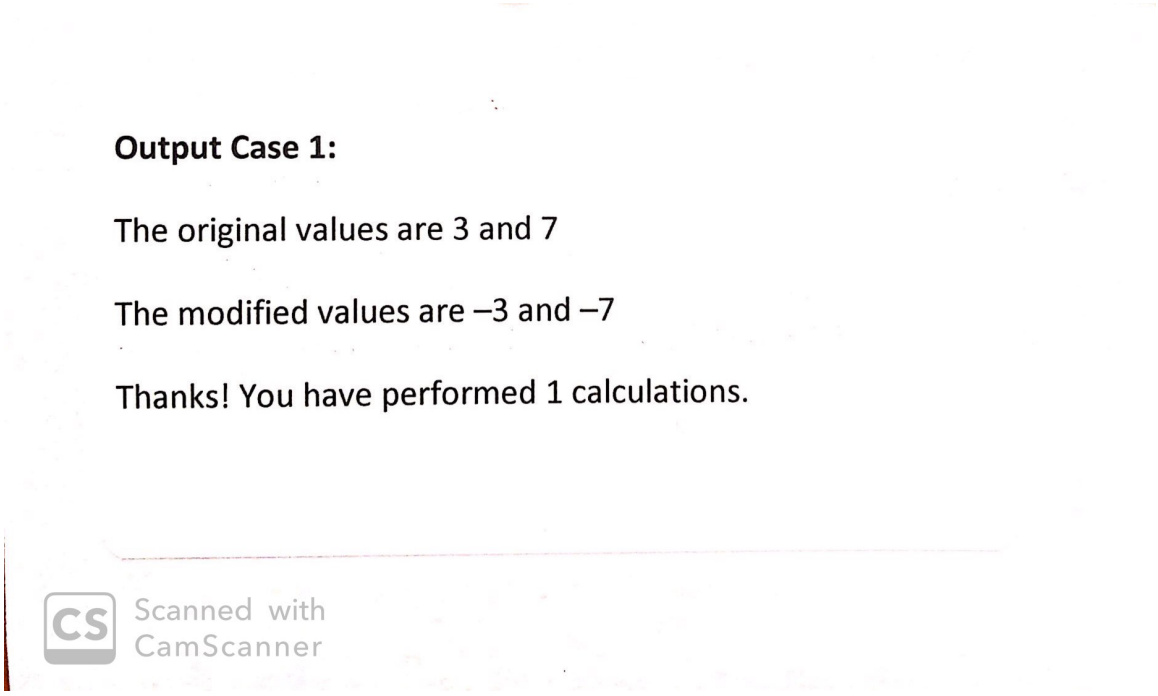

Figure 7.5: HAG Output

Chapter 8

Evaluation

In this study, I have obtained both quantitative and qualitative data and employed a mixed-methods analysis approach, as shown in Figure 8.1 and described in more detail later in the chapter. First, qualitative methods were used to identify misconceptions in the students' EYR pre-tests and post-tests. Then, quantitative data was gathered in the form of counts of these observed misconception counts. Although I collect and analyze this data, the context of the observed learning outcomes was determined based on qualitative analysis of the conversations students participated in during the interventions, using the language exhibited during the MBF intervention or HAG protocol to explain the transition from pre-test EYR explanations to post-test EYR explanations. This chapter describes the mixed method analysis used for this study, the results, discussion of the results, and ends with my limitations, conclusions, and final contributions.

8.1 Analysis

8.1.1 Identifying misconceptions using closed coding

To determine the misconceptions exhibited by study participants, "explain your reasoning (EYR)" pre-tests and post-tests were created as seen in Chapter 4. These questions allowed for students to exhibit misconceptions on the concepts of interest. Figure 8.2 shows a sample EYR test question involving Pointers and PBR vs. PBV. Line 6 starts the **main** function block. Lines 7 and 8 declare two variables, setting **grape**'s value to -4 and **banana**'s value to 6. On line 9, functionG

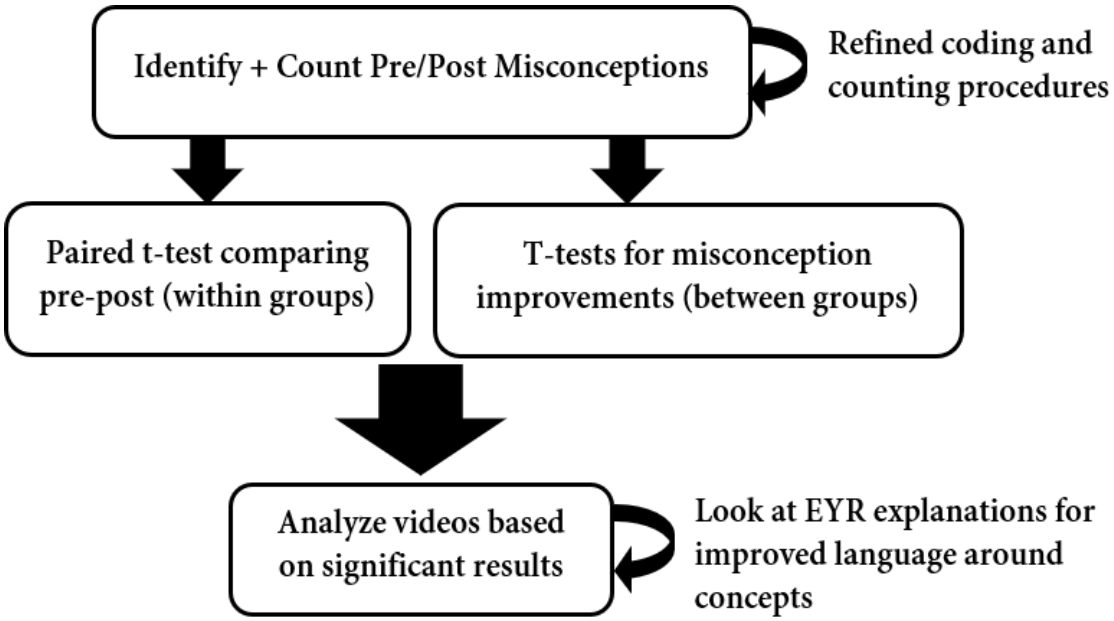

Figure 8.1: Analysis Order

is called, which passes in the address of **grape** (a pointer) and the value of **banana**. After line 6, the function moves to line 1, where we see a void function (does not return anything) declared that takes in two parameters, one integer pointer **a** and one integer variable **b**. Line 2 then performs manipulation on **a** by dereferencing the pointer to obtain its value and then subtracting the value of **b** from it and assigning that result to *a. Since this manipulation is done using a pointer, it is PBR. Line 3 also performs a manipulation, this time on **b**. **b** is multiplied by itself, or effectively squared. This manipulation uses PBV. The correct answer to the values of **grape** and **banana** after functionG returns are **grape** = -10 and **banana** = 6, since **banana** is PBV and **grape** is PBR. Below I provide examples of how the closed coding occurred for the misconceptions exhibited in the study. In Figures 8.3 and 8.4, I provide a summary for all 8 questions of the concepts they focus on, the misconceptions exhibited, and examples of the language seen in the EYR tests.

Sample Explanations:

In referring back to the table of misconceptions in Table 7.1, the question shown in Figure 8.2 had students exhibit misconceptions surrounding PBR such as "No return for functionG to affect values" (ID 105). This explanation showed a PBR misconception, and more specifically Pointers Not PBR, as seen in Table 7.1. The student further exhibited the misconception that the function is not PBR because it is a void function ("No return.")

Q7: Consider the code segment below:

```
01. void functionG (int* a, int b){
02.     *a = *a - b;
03.     b = b * b;
04. }
05.
06. int main( ) {
07.     int grape = -4;
08.     int banana = 6;
09.     functionG (&grape, banana);
10.     return 0;
11. }
```

What are the values of variables **grape** and **banana** after **functionG** returns?

grape:

banana:

Explain your reasoning below:
...

Figure 8.2: A sample test question

Another provided explanation that exhibits both the PBR and PBV misconceptions is "[var1] changes memory location, and [var2] changes the variable's value" (154). This student is confused on the terminology and believes since **banana** is PBV this means that it can change the value of the variable and since PBR involves passing an address, it is the address that is changed and not the value. This student also exhibits the Pointers Not PBR misconception and a Pointers misconceptions, not understanding the semantics of the address and value that are associated with a pointer variable.

8.1.2 Mixed Methods

My study utilizes a mixed methods approach for analysis, as it is necessary to look at both quantitative and qualitative results in an integrated way to paint the full picture of the benefits of the intervention. To be more specific, I utilize first a sequential exploratory analysis design[200], beginning with qualitative analysis and following this up with quantitative. In this form of mixed methods, the qualitative results take precedence. For the purposes of my study, the first qualitative part takes the form of closed coding of misconceptions, described in the "EYR Closed Coding"

Question description	Concept/s	Misconceptions	Examples of Language
Q1: Void function swapping two variables using PBV	PBV	PBV	"temp is set to hold what value is in x or cat. Then x or cat is set to y or dog's value, this changes temp's value accordingly. So now temp holds y or dog's value. Then y or dog is set with the value in temp which has been since set to y or dog's value"
Q2: Int function returning the product/quotient of two variables	Return values	Return Values	N/A
Q3: Void function calculating product/quotient of two variables with same name but different scope	Scope, PBV	Scope; Same name, same location	"The void function doesn't return an integer, so the final result is from the num in the main function."
Q4: Int function calculating sum/difference of two variables and returning that sum/difference, but printing variable in main function with different value and scope	Scope, Return values,	Scope; Same name, same location; Return Values	"value and other value are the same."

Figure 8.3: Descriptions of the first 4 EYR pre/post-test questions

section 8.1.1.1. Following this phase, I performed quantitative analysis by gathering misconception counts and running appropriate statistical analyses to gauge differences in performance/learning between the MBF intervention and the HAG control group, as described in more detail in section 8.2.1. My hypothesis was that the MBF group would have fewer misconceptions on the post-test compared to the pre-test and that the difference of the pre-test to post-test misconceptions would be greater for the MBF group than the HAG group.

After the sequential exploratory analysis, my design pivots and begins a concurrent nested analysis phase[200]. This form of mixed methods has qualitative and quantitative analyses occurring at the same time, but with one taking precedence over the other. In my case, the qualitative takes precedence, being guided by the quantitative to gauge where best to look for qualitative results to provide context for the quantitative results. After tallying the misconception counts, I performed a thematic analysis, a method for identifying, analyzing, and reporting patterns, or themes, within data[15] for the MBF and HAG groups, using the quantitative results as a basis for where to look in the audio transcripts as well as the pre-test and post-test language that were part of determining the themes. This language consisted of student explanations that incorporated vocabulary introduced and discussed during the experimental phase, and exhibiting a more (or less in some instances)

Question description	Concept/s	Misconceptions	Examples of Language
Q5: Void function swapping two variables using PBR and arrays	PBR, Arrays	PBR; Arrays not PBR; Void function not PBR (Arrays)	"The array zebra is not changed by the function, since nothing is returned to the main function"
Q6: Void function calculating the difference/sum and product/quotient of two variables using PBR and pointers	PBR, Pointers	PBR; Pointers not PBR; Void function not PBR (Pointers)	"They remain the same because the function returns nothing"
Q7: Void function calculating the sum/difference of a variable using PBR and pointers and the quotient/product of a variable using PBV	PBV vs. PBR, Pointers	PBR; Pointers not PBR; Void function not PBR (Pointers); Pointers; PBV	""Apple" changes memory location, and orange changes the variable's value"
Q8: Int function manipulating a variable using PBV, manipulating a value using PBR and arrays, and returning an array element	PBV vs. PBR, Arrays	PBR; Arrays not PBR; Void function not PBR (Arrays); PBV	"You return x[0] with didn't change in the function but everything else isn't returned so it remains the same."

Figure 8.4: Descriptions of the last 4 EYR pre/post-test questions

complete grasp of the concepts during the explanation part of the EYR post-tests. This analysis led to a thematic analysis, which consists of six phases, the six being as follows[15]:

1. Familiarizing yourself with the data
2. Generating initial codes
3. Searching for themes
4. Reviewing themes
5. Defining and naming themes
6. Producing the report

For this thematic analysis, the data consisted of the video intervention audio, which was transcribed. In the following sections, I discuss in more detail the analysis methods used for the MBF study.

8.1.2.1 EYR Closed Coding

To analyze the EYR pre- and post-tests, I first transcribed the students' explanations into an electronic form. Each student was given a unique identifier (ID). After the data was transcribed, it was exported into a spreadsheet format. From there, closed coding was done based on a rubric I designed to determine observable misconceptions. The rubric consisted of the misconceptions seen in Table 7.1.

Three coders analyzed the data, with myself coding all of the tests and the other two coding a subset to help ensure consistency of coding. The other two coders had graduate degrees in computer science and were both familiar with the closed coding process and my research and the related work on misconceptions and identifying them. They were not, however, as familiar with my data as I was, so that they could provide outside perspectives. To ensure consistency, the coders started with a rubric that was developed from the misconceptions and difficulties identified in the formative studies in chapters 4 and 5. The coders individually reviewed and blindly coded the data, being unaware of which lab section the student was in (to avoid implicit or explicit bias towards ideal results). Each question response was observed and was given a mark of "1" for a given misconception if the student's explanation indicated that they held that particular misconception about that question. Note that it is possible that student misconceptions were context-dependent – exhibited for some questions but not for others. It was also possible for multiple misconceptions to be exhibited in a single question. The coders would then come together and would discuss discrepancies and explain their thought process behind certain codes. This process was done iteratively and both the codes and rubric were refined as necessary until all the coders came to a consensus about coding the transcripts. This process allowed us to refine our closed coding until our results were identical on a given subset of transcripts. Although we did not calculate a kappa value to determine a precise interrater reliability (IRR)[133], this process was done to ensure that there was a reasonable "extent of agreement among data collectors," as McHugh defines IRR. Once coding was completed for the tests, the IDs were traced back to their section and grouped by section, so that the frequency counts could be analyzed per section.

8.2 Results

8.2.1 Quantitative

To look at the effectiveness of the interventions, various statistical tests to compare means were performed. Table 8.3 shows the changes in misconceptions for each of the 8 questions broken down by group (MBF and HAG) and by partner (A and B). In this chart, the bolded cells show a statistically significant result, meaning that the group (MBF, MBF B, HAG B, etc.) for a given question (Q1, Q3, Q5, etc.) had significantly fewer misconceptions on the post-test than the pre-test. A negative percentage represents misconceptions being introduced in the post-test that were not observable on the pre-test. Following this chart, I provide in Tables 8.4-8.10 an overview of each question, showing the counts of misconceptions on the pre-tests, the post-tests, and then the percent changed, with misconceptions reduced being positive values and misconceptions gained being negative values.

To observe the effects of the interventions compared to one another, t-tests were conducted between groups, comparing MBF to HAG. These tests compared the differences of means, with the type of t-test used depending on whether the variances were equal or unequal, and calculated whether one group performed significantly better than the other on a per question basis. The null hypotheses for these t-tests were that there was no difference between the differences of the pre-test and post-test scores of the two groups (MBF and HAG). The alternative hypothesis is that the MBF group had fewer misconceptions when comparing the differences of the pre-test and post-test scores. Table's 8.11-8.18 show the results of these t-tests, with the first number representing the p-value (0.05 being significant) and the second number representing the difference of means. Bolded numbers represent cases where there was a significant result based on the alternative hypothesis. The general trend of these quantitative results shows that MBF provided benefits for half of the questions, particularly for Q6 and Q7 where significant results are found compared to the HAG protocol. Q6 and Q7 relate to PBV and PBR, as seen in Table 7.1. Observing the data (both in terms of misconception counts and the discussions during the MBF intervention), it seems these questions are the ones students had the most misconceptions about and also the concepts that students spent the most time on discussing using the prompts. It is important to note that although these numbers have positive implications for the benefits of the technique, this study did not have the power to make quantitative claims about the benefits. Although the sections were randomly

Table 8.1: ANOVA per question for Partner As

Question	Condition (MBF vs. HAG)	Test (Pre. vs. Post)	Test x Condition
Overall	0.22	0.22	0.099
Q1	0.49	0.18	**0.03**
Q2	0.29	0.29	0.29
Q4	0.91	**0.01**	0.22
Q5	0.496	0.34	0.94
Q6	0.20	0.48	0.30
Q7	0.20	0.48	0.21
Q8	0.33	0.17	0.08

assigned to treatment or control, the context of the experiment did not permit randomized selection of individuals, so it cannot be assured that the characteristics of one section were equivalent to the characteristics of the other, apart from a potential posthoc analysis on the grades of the students in the course. Following a mixed methods analysis approach, I observe and analyze qualitative results to provide evidence and context for the quantitative results found.

Mixed ANOVA

To look at the effects from another angle, I performed a mixed analysis of variance (ANOVA). For the purposes of this study, the independent variables were the condition (MBF vs. HAG feedback); the partner (A vs. B); and the test taken (Pre-test vs. Post-test). The dependent variables were the misconception counts. The ANOVA looked for main effects that these independent variables might have on the dependent variables as well as any interaction effects (Test x Condition). The results of the mixed ANOVA tests, shown in Tables 8.1-8.2, have statistically significant results ($p < 0.05$) bolded. The ANOVAs were performed on a by partner basis, so Table 8.1 shows the results for Partner As and Table 8.2 shows the results for Partner Bs.

The ANOVA data mostly supports the results of the t-tests, although based on the ANOVA data, the significant improvement for MBF over HAG on Q7 is only true if the participant was Partner B. When looking further into the qualitative results, there is support for Partner A having improved learning outcomes as well, but considering the low number of misconception counts that the Partner As had on the pre-test, the improvement was not statistically significant when running an ANOVA on the data. Having a larger sample size to run these calculations on would potentially help improve the quantitative results.

In the Question Overview tables, numbers in parentheses represent the n (sample size) for the group. The letters (A or B) represent whether the student was providing feedback (A)

Table 8.2: ANOVA per question for Partner Bs

Question	Condition (MBF vs. HAG)	Test (Pre. vs. Post)	Test x Condition
Overall	0.87	0.08	0.93
Q1	0.33	0.90	**0.004**
Q2	0.25	0.58	0.58
Q4	0.50	0.16	0.16
Q5	0.51	0.57	0.57
Q6	0.83	**0.03**	0.36
Q7	0.23	**0.049**	**0.03**
Q8	0.80	0.44	0.81

Table 8.3: Change in Misconceptions by Group

Questions	MBF n=26	MBF A n=13	MBF B n=13	HAG A n=23	HAG B n=27
Overall	**33%**	**38%**	27%	-5%	24%
Q1	-36%	-13%	-67%	**44%**	**53%**
Q2	0%	0%	0%	0%	0%
Q4	**80%**	**100%**	0%	56%	**77%**
Q5	18%	25%	14%	20%	-20%
Q6	**100%**	**100%**	**100%**	-14%	54%
Q7	**74%**	**67%**	**77%**	-15%	10%
Q8	0%	11%	-11%	-87%	-25%

or receiving it (B). These tables break down the number of misconceptions in the pre-tests and post-tests, the percent changed, and the results of a paired t-test. For the paired t-tests, the null hypothesis was that the intervention (MBF) or control (HAG) resulted in maintaining the same number of misconceptions in the post-test as the pre-test. The alternative hypothesis was that there were fewer misconceptions on the post-test than the pre-test, with the expectation that learning had occurred. In Tables 8.4-8.10, the instances that show a significant p-value (<0.05) are shown with the difference of means shown in the same cell to give insight into effect size and are bolded. P-values of less than 0.05 with negative difference of means are not bolded as they are not significant based on the alternative hypothesis. This was done because I was not looking for statistically significant negative outcomes, as students were expected to either remain the same from a control protocol (HAG) or improve from an intervention to help learning (MBF).

In the next section, I discuss the qualitative data found by analyzing the audio and screen captures from the interventions and the language of the pre-tests and post-tests.

Table 8.4: Q1 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value Difference of Means
MBF (26)	14	19	-36	0.03/-0.19
MBF A (13)	8	9	-13	0.17/-0.77
MBF B (13)	6	10	-67	0.05/-0.31
HAG A (23)	16	9	44	**0.008/0.31**
HAG B (27)	17	8	53	**0.005/0.33**

Table 8.5: Q2 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	0	0	NA	NA/0
MBF A (13)	0	0	NA	NA/0
MBF B (13)	0	0	NA	NA/0
HAG A (23)	0	0	NA	NA/0
HAG B (27)	0	0	NA	NA/0

8.2.1.1 Discussion

It is worth noting that in Table 8.3, some questions either do not show significant differences (Q2, Q5, and Q8) or show the HAG group outperforming the MBF group (Q1). The statistically significant quantitative results gave insight into where to look in the qualitative analysis to provide context, but the resulting analysis also gave context for the results that did not show the benefits of MBF under certain conditions. With respect to Q1, the qualitative evidence did not support the HAG protocol reducing their misconceptions nor the MBF group for gaining misconceptions. When examining Q1 and Q5, which both deal with the use of a swap algorithm, the results show the MBF group performing slightly but not significantly better than the HAG group on Q5 and the HAG group performing better than the MBF group on Q1. The qualitative analysis of the two groups support that for questions relating to the swap algorithm, students encountered a conceptual difficulty beyond whether the code used PBV or PBR semantics. Students seemed to stick with a heuristic of either the swap algorithm always swapped the values (behaving in a PBR manner) or they never swapped the values (PBV in nature). On Q2, no misconceptions were seen on either the pre-test or the post-test, so the percent changed is not applicable. For Q3, the post-test question was designed in a way that did not allow for the testing of the same concepts/look for the same misconceptions (namely Scope and Same Name, Same Location), so it was not included for analysis. This occurred as a result of an unintentional flaw in the experimental design and explains why the

Table 8.6: Q4 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	10	2	80	**0.04/0.31**
MBF A (13)	8	0	100	**0.04/0.67**
MBF B (13)	2	2	0	NA/0
HAG A (23)	9	4	56	0.08/0.22
HAG B (27)	13	3	77	**0.02/0,37**

Table 8.7: Q5 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	11	9	18	0.08/0.08
MBF A (13)	6	4	33	0.08/0.15
MBF B (13)	5	5	0	NA/0
HAG A (23)	15	12	20	0.27/0.13
HAG B (27)	15	18	-20	0.21/-0.11

reductions for this questions are noticeably higher than the other questions, so the question's results are not included. The other questions (Q4 and Q8), show a lessened version of expected results, with the MBF groups outperforming the HAG groups. However, this performance was not significant, potentially because as found in the qualitative analysis, multiple MBF groups went through the Pointers prompts but did not go as in depth through the Arrays prompts, and Q8 was based on arrays. Q4, which focused on scope misconceptions, was also found in the qualitative analysis to not be discussed with as much detail in the MBF groups and also did not have as many misconceptions in the pre-test as some of the other questions, which may contribute to why it did not show statistical significance. Overall, an observation was that a number of groups were not compliant with the expected protocols (MBF or HAG). This led to HAG groups having some benefits from the human side-chats versus pure autograder feedback and some MBF groups not having benefits because they did not engage in discussions as the intervention was designed.

Table 8.8: Q6 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	11	0	100	**0.01/0.42**
MBF A (13)	3	0	100	**0.04/0.23**
MBF B (13)	8	0	100	**0.04/0.62**
HAG A (23)	7	8	-14	0.41/-0.04
HAG B (27)	13	6	54	0.11/0.26

Table 8.9: Q7 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	19	5	74	**0.008/0.54**
MBF A (13)	6	2	67	0.13/0.31
MBF B (13)	13	3	77	**0.02/0.77**
HAG A (23)	13	15	-15	0.31/-0.09
HAG B (27)	10	9	10	0.43/0.04

Table 8.10: Q8 Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
MBF (26)	18	18	0	0.5/0
MBF A (13)	9	8	11	0.40/0.08
MBF B (13)	8	9	-11	0.40/-0.08
HAG A (23)	15	28	-87	0.006/-0.57
HAG B (27)	16	20	-25	0.16/-0.15

Table 8.11: Q1 T-Tests and Difference of Means

	MBF (26)	MBF A (13)	MBF B (13)
HAG A (23)	0.0009/-0.50	0.008/-0.30	0.02/-0.67
HAG B (27)	0.0006/-0.56	0.005/-0.33	0.001/-0.69

Table 8.12: Q2 T-Tests and Difference of Means

	MBF (26)	MBF A (13)	MBF B (13)
HAG A (23)	NA/0	NA/0	NA/0
HAG B (27)	NA/0	NA/0	NA/0

Table 8.13: Q3 T-Tests and Difference of Means

	MBF (26)	MBF A (13)	MBF B (13)
HAG A (23)	0.48/0.01	0.39/0.10	0.43/-0.07
HAG B (27)	0.41/0.06	0.33/-0.15	0.48/-0.02

Table 8.14: Q4 T-Tests and Difference of Means

	MBF (26)	MBF A (13)	MBF B (13)
HAG A (23)	0.35/0.09	0.13/0.45	0.09/-0.22
HAG B (27)	0.40/-0.06	0.21/-0.30	0.02/-0.37

Table 8.15: Q5 T-Tests and Difference of Means

	MBF (26)	MBF A (13)	MBF B (13)
HAG A (23)	0.34/-0.09	0.42/-0.05	0.29/-0.31
HAG B (27)	0.17/0.15	0.18/0.19	0.26/0.11

Table 8.16: Q6 T-Tests and Difference of Means

	MBF (26)	**MBF A (13)**	**MBF B (13)**
HAG A (23)	**0.04/0.47**	0.15/0.27	**0.05/0.66**
HAG B (27)	0.28/0.16	0.49/-0.009	0.20/0.31

Table 8.17: Q7 T-Tests and Difference of Means

	MBF (26)	**MBF A (13)**	**MBF B (13)**
HAG A (23)	**0.01/0.63**	0.11/0.39	**0.02/0.86**
HAG B (27)	**0.04/0.50**	0.20/0.30	**0.03/0.73**

Error and Considerations Multiple t-tests were conducted for a portion of the quantitative analysis. Thus, there is a possibility of type 1 Error, or results showing a false-positive, which would show a significant result when the results were not significant. This would be the case for the per-question statistical analyses. When a Bonferroni correction[212] is applied, considering only 7 of the questions were valid for analysis, the adjusted p-value becomes 0.05/7, or 0.007. Using this new p-value, no statistically significant results arise per question, with the closest being the within groups paired t-test for Q7 having a p-value of 0.008. Some research has deemed attempting to correct for multiple t-tests with a Bonferroni adjustment as unnecessary[171]. Rothman states that the adjustment meant to minimize type 1 error will increase type 2 error, or results that do not reject the null when they should, known more commonly as "false-positive." In a quote from this work, it is said "A policy of not making adjustments for multiple comparisons is preferable because it will lead to fewer errors of interpretation when the data under evaluation are not random numbers but actual observations on nature."[171] The Bonferroni correction is also potentially over correcting as although multiple t-tests are performed, what is being tested for is different question to question, as each question represents different concepts.

Another consideration was heteroscedastity, or whether or not the variances of the groups were equal. To account for this, an F-test was performed on the data sets before each between groups t-test. For this F-test, the null hypothesis is that the two variances are equal and the alternative

Table 8.18: Q8 T-Tests and Difference of Means

	MBF (26)	**MBF A (13)**	**MBF B (13)**
HAG A (23)	**0.03/0.57**	0.06/0.57	**0.05/0.57**
HAG B (27)	0.28/0.15	0.31/0.15	0.30/0.15

is that they are unequal. Using a software to perform the F-test, the results provide an F-value and F-critical value. If the F-value is greater than the F-critical value, you fail to reject the null and if the F value is less than F-critical, you reject the null hypothesis that the two variances are unequal. These results then allowed me to perform the appropriate t-test (assuming equal variances or assuming unequal variances) depending on the F-test performed.

Legitimation Framework As this process of analysis was extensive, I needed to consider how to ensure quality for these analysis methods. To do this, I look to the legitimation framework as described in[143]. For this study, one of the relevant forms of legitimation is weakness minimization, which is defined as "the extent to which the weakness from one approach is compensated by the strengths of the other approach." The 'approaches' in this case are quantitative and qualitative analysis methods. For the MBF evaluation, the strengths of the qualitative analysis compensate for the weaknesses of the quantitative. Although there were statistically significant quantitative results found to support the benefits of MBF, the qualitative analysis allows for a deeper explanation of the context of these results and allows for broader claims to be made. Another legitimation that was considered was sequential[143]. This played a role in the mixing of data, as the order of how the phases (quantitative and qualitative) of data are collected can play a factor in the interpretation. Considering this, a concurrent nested[200] approach was taken for the thematic analysis, having qualitative and quantitative analyses occurring at the same time, but kept separate and not informing each other. The data were then mixed and integrated when time to interpret the data. To additionally reduce sequential legitimation threat, there were waves of data analysis performed in different orders, with additional statistical analyses performed on the data after conducting a qualitative phase. In the next section I discuss how the screen and audio recordings of the two conditions (MBF and HAG) were analyzed and the results that came from the analysis.

8.2.2 Qualitative Analysis of Videos

By categorizing student discussions in this framework while also looking for evidence in the pre/post-test explanations, I was able to group certain categories of MBF partners and provide common themes for the learning gains exhibited. Based on these categories, I performed a thematic analysis[15], as described in 8.1.1 on the qualitative data and provide themes for these four categories. The four MBF categories that developed from this process were:

- **Non-Conversational:** These were pairs of students who did not actively engage in dialogue. Usually, this meant that **B** only responded to the prompts presented by **A** and **A** offered no additional dialogue.
- **Conversational (Off-Prompt):** These were pairs of students who were engaging in dialogue, but the dialogue was not based on the provided prompts. Usually, this meant the pair was discussing how to fix **B**'s code without discussing the concepts the MBF prompts were designed to address.
- **Conversational (Semi On-Prompt):** These were pairs of students who engaged in dialogue, but did not follow the prompts in the prescribed order. They used the prompts to provide more fodder for questions or when relevant, but also have a substantial amount of conversation about solving the problem without utilizing the prompts.
- **Conversational (On Prompt):** These were pairs of students who engaged in dialogue centered around the provided MBF prompts. These students followed the path of discussing the prompts and concepts as they were presented, regardless of if they had successfully finished the program before the intervention.

This section walks through the results that came up through a qualitative analysis of the audio and screen capture of the pairs engaging in the interventions. The results are presented as video observations first, and then the language changes noticed in the pre-tests and post-tests. Multiple instances occurred where the MBF and HAG groups use language in the intervention/protocol or the pre-test/post-test that directly supports students having improved learning outcomes or students gaining misconceptions. These are reported and discussed throughout the results section. For both the MBF and HAG groups, cases are seen where no direct evidence supports either the improved learning outcomes or students gaining misconceptions between the pre-test and post-test. In these instances, a variety of explanations are possible. One thought is that the students reviewed material between the two days of the intervention. This could potentially help or hurt them, depending on if their studying led to a correct understanding of the concepts. Another possible explanation for learning benefits is that testing experience and environments themselves caused students to learn. Being provided the pre-test and then the opportunity to implement a solution through coding may have caused the students to think about the concepts between the two days of the study

without actively reviewing the material. For the HAG pairs, it is also possible that the students benefited from the social interaction or human element present during the protocol and from their interactions outside of the protocol. To tease this apart, a future study could perform the more realistic comparison of a computer autograder directly versus the MBF technique to control for the human element. Another testing effect consideration is that a learning effect occurred, where students were able to recognize the similarities of the questions on the post-test and provide more appropriate and conceptually accurate responses on the post-test. In my results, I mention instances where there is not direct evidence to support the observed learning outcomes. Each entry pair in the results section is framed by the ID number of the student who was **A** (feedback provider) listed first followed by the **B** (feedback receiver). For example, if **A** was 74 and **B** was 203, the entry is listed as **(74-203)** There are some entries where one of the partners is listed as "Unknown." These cases occurred when one of the partners did not complete either a pre-test, post-test, or both, so it was not possible to perform analysis on the data for these individuals. I present the MBF entries in the four themed categories that emerged from the thematic analysis. At the beginning of each category, I have provided a table that gives an overview of the observed qualitative results. For the MBF groups, this table has the following columns:

- **ID:** This column provides the ID number randomly generated for the student.
- **Misconceptions introduced:** A checkmark indicates that misconceptions were introduced in the post-test that did not exist in the pre-test.
- **Misconceptions reduced:** A checkmark indicates that misconceptions that existed in the pre-test were not observed in the post-test.
- **Improved Language:** A checkmark indicates that the language used on the post-test showed improvement from the pre-test. In most cases, this aligns with **Misconceptions reduced,** as misconceptions were coded based on the pre-test and post-test language. However, there are a few instances where students did not exhibit observable misconceptions in the pre-test and also did not exhibit misconceptions in the post-test, but the language on the post-test improved. Improved language occurred when students: now use terminology/vocabulary correctly that was not used or used incorrectly on the pre-test; correctly explain a concept that was incorrectly explained in the pre-test.

- **Support:** A checkmark indicates that there is evidence in the audio recordings during the intervention to support the improved language/reduced misconceptions noted for the student.
- **Code working:** This is only relevant for **B**, as they were the students to bring up their code during the intervention. A checkmark + indicates the student started the intervention with code that compiled, executed, and used PBR to implement the solution. A checkmark indicates the student started the intervention with code that would compile and run, but did not fully have the functionality required or was implemented not following the instructions. No checkmark indicates that the student came into the intervention without having working code.

8.2.2.1 Non-Conversational MBF:

A					B					
ID	Misconceptions introduced	Misconceptions reduced	Improved language	Support	ID	Code working	Misconceptions introduced	Misconceptions reduced	Improved language	Support
85			✓	✓	124	✓+				
160		✓	✓	✓	254					
393					294					
398					350	✓				
276					N/A	✓	N/A	N/A	N/A	N/A

Figure 8.5: Overview of MBF Non-Conversational

(276-Unknown) Video Observations: This pair begins with **B** having working code using PBV to implement a solution. In this instance, the partners are not being conversational and instead **A** is asking the questions on the prompts and **B** is replying to them. **B** does not have detailed comments about the concepts represented in the prompts.

Language in Pre/Post-tests: There is no change in language for either partner in this instance.

(398-350) Video Observations: In this pairing, **B** started out with code that compiled and ran but did not follow instructions. He states "In my program, I just used one function in the main, so I didn't actually use a flip function." **B** begins discussing pointers, focusing on how they theoretically could have used them, saying things like "I didn't use pointers but if I were to use pointers, I would

have needed to use them if I created a flip function on its own." and "I would have called that flip function and in order to use that flip function I would have had to use pointers." When asked about whether pointers would pass the address or the value, **B** responds "You pass the address of a pointer. You pass the address of the value, not the actual value." **B** then continues with a misunderstanding of PBV, stating "Because if you pass the value, the value can be manipulated." **B** seems to have PBV and PBR confused, saying about pointers "Well if I had used a separate function and used pointers, the value should not be manipulated or changed, but if that were to happen, it would have been because I passed in the actual values instead of the addresses." Interestingly, **B** believes there is a difference between pass by pointer (a non-standard term) and pass by reference, asking "Pass by pointer passes in the address and then pass by reference passes in the actual value, right?" This shows either confusion in terminology or potentially discussing concepts that are not generally taught in CS1 courses. The pair continues throughout the prompt with little conversation and **B** not providing any more in-depth replies.

Language in Pre/Post-tests: This pair shows no notable language differences from pre-test to post-test.

(85-124) Video Observations: This pair begins with **B** having working code that is implemented using pointers. When prompted about whether values are returned from the flip function, **B** correctly responds "Values are not returned using a return statement but they're stored using pointers." Discussing how the flipped values are accessed, **B** states "The flipped values are accessed by taking the values and using pointers to carry over the signFlipper variables back into the main function and then it displays it." **B** continues to respond to the prompts and explains the use of pointers in their solution. There is not much conversation and **B** provides straightforward answers.

Language in Pre/Post-tests: In the pre-test for Q6, **A** comments "Uh I lowkey don't know what happened." They state that they just did the math, unsure of the concepts why it worked. In the post-test for Q6, **A** explains correctly "*a is a pointer back to the main function."

(160-254) Video Observations: This pair begins with **B** having code that does not compile and originally attempts to return two integer values from a function. **A** does question this at a point, asking "How would you? Because you're returning two values?" to which **B** discusses a workaround of "Well what I would do is just be stupid and write two functions." **B** follows this by mentioning PBR, saying "But if I was to... If I needed to return two, I would just use pointers and arrays to

return." When the MBF prompts specifically ask about PBV or PBR, **B** states "Well if I correctly did the program, I would have used a...Pass by reference" because "Pass by value doesn't change the main block numbers." This consistent thought process continues when asked whether an address or value was passed, with **B** echoing "If you pass it by value, then only the numbers in the function block change...But if you want to change the value after the function call, you would have to use pass by reference." When the pair begins with the Arrays prompts, there seems to be confusion from **B**, particularly with arrays operating in a PBR-like manner. After being asked if the values in the array are different after the function call returns, **B** responds "This one I'm not sure because I don't think the value changes.." and follows this up with "I would have used pointers." Towards the end of the prompts, **B** is asked how assigning a value to a variable in one function would affect a variable of the same name in a different function. **B** correctly replies with a solid grasp of the concept of scope "Umm, they're completely separate because they're two different function blocks. So whatever happens in one doesn't carry over to the next function block."

Language in Pre/Post-tests: In the pre-test, **A** originally remarks for Q6 that they are "Not sure what * does, cow = 13 - 6 = 7 duck = 10 * 2 = 20." This shows evidence of just doing the math and not understanding the concept/syntax of pointers. After the intervention, on Q7 (also about pointers), **A** states that "Grape would be equal to (-4 - 6) 10" but "banana would stay 6 because there is no pointer in the functionG," showing an understanding of how to use/interpret the syntax of pointers. Another noteworthy language change for **A** appears in Q4, where on the pre-test, it is said that "Number is returned as (8-5) 3 and other_number = number in line 10," an incorrect use and understanding of scope. In the post-test, **A** on Q4 says "Value is never changed from -5 and other_value is equal to 2 from functionD." Although not a language change, it is noted that **B** on the post-test seems to believe that arrays are not PBR, which supports the confusion exhibited during the intervention. This confusion was never challenged or discussed further because this pair was not conversational.

(393-294) Video Observations: This pair begins with **B** having code that doesn't compile, stating "Yeah. This doesn't work at all." When asked about the flow of control of the program, **B** correctly states "Well pretty much you need to have a function that uses pointers to change the signs. Because you can't return two values from a function." One of the few interactions between the pair comes when for void functions, **B** states "Yeah, but don't you still have to add a return

statement?" and **A** wonders "Do you have to?" **B** suggests that they compile and test it. After spending some time working to fix the code, **B** realizes that the code was not saved as a ".c file" and addresses that issue. When prompted about the return type of the function, **B** recalls and says "We're gonna test our theory to see which one is right." After testing, **B** accepts that **A** was right that the return statement is not required, saying "Well if you write return or don't write return in a void type function, it's not going to matter." The pair discusses pointers when asked about if there are values returned from the flip function. **B** states "So pretty much all you need to do is use the pointers and then it's going to change the values in the memory allocated by the pointers." After being prompted about whether PBV or PBR was used, **B** comments "I'd say we probably used pass by reference...Because of using pointers to go through and then adjust them with the pointers so you could keep on going through." During the Pointers prompts, **B** discusses in depth how the pointers go through the code, even mentioning "And our flipSign function is expecting pointers and so it all works out because the types are the same." The terminology of dereferencing confuses **B**, with the incorrect thought of "I know dereferencing is usually when you put an ampersand in from of your pointer" (Dereferencing is putting a star "*" in front of the pointer variable). However, after the MBF prompt asking whether an address or value was passed for pointers, **B** states that an address was passed and "If we passed the value it wouldn't have done anything because you change the value inside the function and then it would just go back to normal because it wouldn't carry over everything because we have a void return type for our function." Explaining this out loud causes **B** to want to return to the dereferencing question, where they add "We dereference the pointers inside our flip function when we put the asterisks in front of our pointers for value x and value y." The pair continues through the prompt but does not go through the Arrays cards.

Language in Pre/Post-tests: Although **B** provided in depth explanations, particularly around the concept of pointers, there were no notable language differences as both partners correctly answered and explained the questions surrounding pointers (Q6 + 7) in both the pre-test and the post-test.

8.2.2.2 Conversational (Off-Prompt) MBF:

(75-142) Video Observations: This pair begins with **B** having code that doesn't compile. There is little in-depth discussion from this pair. Towards the end of the prompts, they begin a discussion about scope and variables of the same name in different scopes. **A** remarks "Didn't she talk about this in class? When she was saying having two of the same thing kind of changes the output of the

A					B					
ID	Misconceptions introduced	Misconceptions reduced	Improved language	Support	ID	Code working	Misconceptions introduced	Misconceptions reduced	Improved language	Support
75					142					
N/A	N/A	N/A	N/A	N/A	1					

Figure 8.6: Overview of MBF Conversational (Off-Prompt)

other one." **B** quickly shoots down this idea, saying "No that was only.. That was only argc and argv."

Language in Pre/Post-tests: This pair shows no notable changes in language from their pre to post-tests.

(Unknown-1) Video Observations: This pair begins with **B** having code that does not compile, remarking "See that's the thing, I don't have the exact code solution." The pair spends time attempting to fix the program. They begin to operate in a pair programming dynamic, with **A** catching syntax issues "Oh make sure you put the parentheses." They spend time discussing the algorithm of how to flip the signs, with **B** mentioning the use of the absolute value function and conditionals. After a while, **A** attempts to see if **B** wants to compile, to which **B** replies "Umm, not yet... which is usually horrible advice." The pair spends time trying to figure out if it's possible to return two values from an integer function, and never get the code working as intended. They also do not spend much time talking about the MBF prompts, instead just attempting to fix the code.

Language in Pre/Post-tests: With this pair, the only language improvement noticed is that **A** mentions "pointers" in the post-test for Q6 and Q7 when the term was not used in the pre-test. However, the pre-test exhibited no misconceptions and based on analysis of the transcript, the only support for this improvement the intervention suggests is **A** having a deck of cards that mentions the term "pointers" frequently.

8.2.2.3 Conversational (Semi On-Prompt) MBF:

(119-154) Video Observations: This pair begins with **B** having code that compiles and runs, but is implemented using PBV. In this instance, it is observed that **B** uses both feedback from **A** and online searching resources to improve. While discussing the prompts during the intervention, **B**

A					B					
ID	Misconceptions introduced	Misconceptions reduced	Improved language	Support	ID	Code working	Misconceptions introduced	Misconceptions reduced	Improved language	Support
119					154	✓ (Attempts PBR)		✓	✓	✓

Figure 8.7: Overview of MBF Conversational (Semi On-Prompt)

brings up a search engine and looks up "first element array pointer." This occurs after remarking "Well arrays and pointers aren't they the same thing, right? Or the first element's a pointer or something?" (An array's name is a pointer to the address of that array's first element). When referring to the functions that "don't return anything," **A** remarks "It actually changes the memory? I guess that's what it does?" This supports discussion of pass by reference and how it is referring to memory locations. The two partners go back and forth on whether or not an array is a pointer or pointing to something, with **A** stating "Well I don't think it's pointing anywhere yet" and "But we haven't like assigned any pointers yet, I don't think." **B** holds fast to this heuristic of "I just know the first element of an array is a pointer."

Language in Pre/Post-tests: With respect to language from pre- to post-test, **B** begins with the incorrect belief that "with pointers the thing that changes is their memory location not the actual value" on Q6. For Q6 in the post-test, the explanation states "The value of the memory address gets changed in the function," now showing an understanding that there are both a value and memory address associated with pointer variables. This participant does not use pass by reference correctly in Q7, mentioning that "[variable] is passed by reference," where the variable is actually passed by value in that particular question.

8.2.2.4 Conversational (On Prompt) MBF:

(301-27) Video Observations: This interaction begins with **B** having code that does not compile, remarking that they "don't even remember what this does." Despite this, the pair begins a discussion using the prompts. When discussing how values are passed, **A** states "Using ampersand. That's pass by value," which is an incorrect statement as passing values using an ampersand is actually pass by reference (PBR). This is another pair that begins interacting as if they were pair programming.

A					B					
ID	Misconceptions introduced	Misconceptions reduced	Improved language	Support	ID	Code working	Misconceptions introduced	Misconceptions reduced	Improved language	Support
105		✓	✓	✓	344					
192		✓	✓	✓	167	✓+				
253					70			✓	✓	✓
301					27	(Yes by end)	✓(Swap Algorithm)			
321					354		✓	✓	✓	✓

Figure 8.8: Overview of MBF Conversational

B has attempted to implement the program incorrectly, saying "Yeah. I return two values from flip." **A** suggests that they "Recompile it" after wondering "In C, how does it return.." confused by the attempt to return two integers. Later, it is discovered that **A** implemented the solution using pointers, causing **B** to inquire "But when you return is it?" **A** informs **B** that there is nothing returned and pointers "just reassign the value." **B** seems to continue to struggle with pointers and eventually **A** decides to avoid the discussion about those cards, remarking "Yeah. Don't even worry about that one." They spend the rest of the time attempting to implement the solution using arrays, ultimately successfully completing the task.

Language in Pre/Post-tests: This pair shows no notable changes in language from their pre to post-tests.

(192-167) Video Observations: This pair begins the intervention with **B** having code that compiles and runs using pointers. During the intervention, it is noticed that some of the language is confusing for **B**, saying "I don't know what that means.." after being asked whether they used pass by value, pass by reference, or something else. **A** helps by responding "Did you pass the pointer by value or by reference. Like did you use its location or use its value?" This causes **B** to remark "I think by reference." **A** helps assure **B**, stating "Oh yeah. You used its location, so reference." Interestingly, a little later, when asked if an address or value was passed, **A** incorrectly responds for **B** saying "You pass a value," even though **B** used pointers and so passed an address into the function. When this pair gets to arrays, it comes to light that **A** implemented the solution using

arrays. The pair decides to continue the discussion with **A** responding to the array questions based on how they implemented the solution.

Language in Pre/Post-tests: **A** in the post-test for Q6 states "Pass by reference allows the functionF to perform its calculations." Although the partner correctly answered the question in the pre-test and exhibited no misconceptions, there was no mention of pass by reference, which supports that the discussion occurring in the intervention helped with that terminology. Similarly for **B**, there is a mention of pointers in the post-test that does not exist in the pre-test. **A** also exhibits confusion over pass by value in the post-test, correctly stating in Q7 saying "a or grape is passed by reference while b or banana is passed by value," and leaving banana unchanged because it is passed by value. However, on Q8, **A** states "Pass by value means that the variables lose their old value to store a new value," and on Q1 remarks "The two variables are swapped using pass by value (I think...)" This confusion is consistent with the conversation present during the intervention.

(253-70) Video Observations: In this situation, **B** begins with code that does not compile. **A** seems to have a better grasp of pointers and through the prompted discussion, **A** is able to improve **B**'s understanding of the concepts. **A** asks if **B** would use pass by value or pass by reference if trying to implement the code using pointers. **B** responds "Probably value?" to which **A** soon says "I feel like you should pass by reference... Cause then you can just change the value of it." After working on the code and making adjustments to incorporate pointers into the implementation, **A** asks the prompt question "How are the addresses and/or values changed inside the function?" To this question, **B** replied "Well, since we're passing the address, it's changing the value at that address?" and **A** confirmed that this is the case.

Language in Pre/Post-tests: Looking at the language used on the EYR pre- and post-tests for Q6 and 7, **B** in this case was uncertain about pointers, remarking "Passing these values by pointer has no actual impact on their original value (or I have it flipped in my brain and it's the opposite)" on the pre-test. After going through the intervention, the post-test had more certainty about the subject, the explanations on the post-test say "functionF calls pig and fox by reference to change the values at their addresses" for Q6 and "grape is called by reference and functionG changes the value at that address. banana is called as a regular integer and, because there is no return on the function, banana stays the same" for Q7.

(321-354) Video Observations: This pair begins with **B** having code that compiles and runs using PBV and calling the function twice to flip the numbers as opposed to using PBR. The audio shows rich discussion from both partners, particularly around the concepts of pointers and how they are PBR. Originally, when they get to the Pointers cards, **A** asks "Well where could they have been created, I guess?" In response to this, **B** responds "So you can use a pointer if you want to just pass the address of wherever the number is stored in your function." This comment shows a basic understanding of what was happening with pointers, but after realizing the rest of those cards were about pointers and that **B** did not implement the solution using pointers, the pair decides to skip over the cards. After finishing the rest of the cards, they inform the researcher that they had skipped the sections irrelevant to their implementation choice. They are then informed to still go through all of the cards and discuss the concepts if they had been implemented that way. This causes the pair to go back and have more in depth discussion about pointers, with **B** getting the opportunity to self explain that "So pointers point to the address. Like the location of wherever that value is stored. So essentially it's passing by reference." Going back to discuss allows **B** to understand that there's a value associated with a pointer's address, stating "So any time you need the value or need to rewrite the value you would have to use the asterisk." The arrays discussion focuses more on syntax than the concepts.

Language in Pre/Post-tests: **B** in the pre-test for Q7 remarks "3. cow/duck not declared as pointers $->$ 13, 10 are memory locations 4. Oof I don't remember a lot about pointers I was banking on a refresher 5. Let's assume %'s are memory locations $->$ only reallocated w/o ok." However, after going through the intervention, for Q6, **B** mentions "passed as pointer" and for Q7 correctly notes that "banana/b doesn't change (not pointer) grape/a changes."

(105-344) Video Observations: In this case, **B** begins with code that does not compile. There seems to be almost a learn through teaching experience for **A**. After prompting **B** asking what the pointers are pointing to, **A** responds saying "Well they're just kind of pointing to... kind of the address of where the data is stored.." **B** has some confusion, questioning "Yeah. Can't you control whether it's set to the address or the value?" **A** then explains things, stating "Well the value is..What goes in the box it's pointing to." **A** appears to have a better grasp on pointers than **B**, and through helping explain how the idea of pointers and them referring to addresses that can hold and alter values, **A** was able to learn. This result back to Chi's self-explanation study[30], with

A getting learning gains from getting to explain the material.

Language in Pre/Post-tests: When referring to pointers during the intervention, **A** remarks "they're kind of pointing to..the address of where the data is stored" and that you "only use the address with pointers."

Even **A**'s explanations in the post-test for these two questions now reference (no pun intended) how the function performs the operations and changes the values "via use of pass by reference and pointers" and noting how for Q7 "It changes banana by pass by value" which would mean the value of the variable named banana would not be changed in the main function.

8.2.3 HAG Pairs:

For the HAG pairs, the categories that developed were as follows:

- **HAGs**: These were pairs of students in which **B** came in with working code and the pair performed the HAG protocol as expected, with **A** only providing confirmation feedback and **B** testing their code.
- **HAG Nots (Irrelevant Discussion)**: These were pairs of students who did not follow the protocol and had discussions relating to programming concepts throughout the intervention, but the discussions were irrelevant to the concepts addressed in the EYR pre-/post-tests.
- **HAG Nots (Relevant Discussion)**: These were pairs of students who did not follow the protocol and had discussions relating to programming concepts throughout the intervention and those discussions were relevant to the concepts addressed in the EYR pre-/post-tests.
- **Other**: This was only one case where the audio was lost a few minutes into the video.

As with the MBF groups, each category begins with an overview of the qualitative results. The columns for the HAG/HAG Nots were as follows:

- **ID:** This column provides the ID number randomly generated for the student.
- **Misconceptions maintained:** A checkmark indicates that the same misconceptions that were present in the pre-test were also present in the post-test.
- **Misconceptions reduced:** A checkmark indicates that misconceptions that existed in the pre-test were not observed in the post-test.

- **Sustained Language:** A checkmark indicates that the language used on the post-test was consistent with the language used in the pre-test. In most cases, this aligns with **Misconceptions maintained,** as misconceptions were coded based on the pre-test and post-test language.

- **Improved Language:** A checkmark indicates that the language used on the post-test showed improvement from the pre-test. In most cases, this aligns with **Misconceptions reduced,** as misconceptions were coded based on the pre-test and post-test language. However, there are a few instances where students did not exhibit observable misconceptions in the pre-test and also did not exhibit misconceptions in the post-test, but the language on the post-test improved. Improved language occurred when students: now use terminology/vocabulary correctly that was not used or used incorrectly on the pre-test; correctly explain a concept that was incorrectly explained in the pre-test.

- **Support:** A checkmark indicates that there is evidence in the audio recordings during the intervention to support either the improved language/reduced misconceptions or the sustained language/maintained misconceptions noted for the student.

- **Code working:** This is only relevant for **B**, as they were the students to bring up their code during the intervention. A checkmark + indicates the student started the intervention with code that compiled, executed, and used PBR to implement the solution. A checkmark indicates the student started the intervention with code that would compile and run, but did not fully have the functionality required or was implemented not following the instructions. No checkmark indicates that the student came into the intervention without having working code.

8.2.3.1 HAGs:

(319-189) Video Observations: This pair begins with **B** having code that compiles and runs. They work through the test cases according to the protocol and successfully complete them all.

Language in Pre/Post-tests: Both **A** and **B** seem to keep the consistent belief that arrays do not operate in a PBR manner for Q8 on the pre-test and post-test. **B** on the pre-test remarks "cow[0], cow[1], and horse do not get assigned/returned new values in functionH..." for Q8 and on the post-test says "shark[0], shark[1], and bear do not change..," showing very similar language and thought

A							B							
ID	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support	ID	Code working	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support
58							335	✓	✓		✓	✓		
59	Gained		Worse				359	✓ (one function)						
94	✓			✓			312	✓	✓		✓	✓		
212	✓		✓	✓			183	(Used Ex.)	✓		✓	✓		
214	N/A	N/A	N/A	N/A	N/A	N/A	147	✓+	✓		✓	✓		
256	✓		✓	✓			35	✓	✓	✓	✓	✓	✓	
319	✓	✓	✓	✓	✓	Swap Alg.	189	✓	✓		✓	✓		
331		✓			✓		141	✓	✓		✓	✓		
N/A	N/A	N/A	N/A	N/A	N/A	N/A	39	✓+	✓		✓	✓		

Figure 8.9: Overview of HAGs

process. **A** on the pre-test for Q8 states "cow and horse are not changed;..." and on the post-test for the same question says "Everything up to val is unchanged,..." Keeping these misconceptions is expected with the pair completing the HAG protocol according to the protocol.

A exhibited changed language not supported by the intervention. For Q5 (arrays and swap algorithm), **A** on the pre-test correctly believes that the values are swapped, saying "The function called in line 11 reverses the order." On the post-test, however, **A** changes the thought process, stating "once function is returned values stay original." A similar change (but from misconception to correct reasoning) is seen on Q1, where **A** on the pre-test incorrectly states "The void function swaps the values for x and y,..." but on the post-test, notices that "void function does not return a value back." **B** similarly has a change of reasoning on Q5 (arrays and swap algorithm), on the pre-test incorrectly stating "In functionE we are simply being passed 4, 7 and then swapping the values in the array x but not returning the array x, so cow[] stays the same," while on the post-test saying "simple swap function like question 1." Analysis of the audio from the intervention provides no evidence that the changes around Q1 and Q5 were related to the intervention.

(58-335) Video Observations: **B** begins with code that compiles and runs and the pair perform the HAG condition as expected, with **A** providing test cases for **B** and responding with whether the outputs were correct or not.

Language in Pre/Post-tests: The language from pre- to post-test supports this as well, where we see in Q6 **B** saying "Values remain unchanged due to the '&' being run through the function" and then in the post-test still believing "Original values of [variables] remain the same."

(256-35) Video Observations: This pair begins with **B** having code that compiles and runs using PBV. After figuring out the instructions, the pair performs the HAG protocol as intended, with **A** only providing confirmation feedback.

Language in Pre/Post-tests: **A** in this case maintains the same misconceptions from pre-test to post-test. In Q1, **A** says on the pre-test "functionA sets the value of int x to the value of int y. x is cat, y is dog." On the post-test for the same question, **A** states "functionA is a value swap function, and it swaps the values of lion & wolf." This is the only notable misconception and the language stays consistent as would be expected from someone performing the HAG duties as intended. **B** maintains pointers misconceptions from pre-test to post-test for Q6 and Q7. On the pre-test for Q6, **B** responds "functionF passes the address of cow and duck so after the function passes, the values haven't changed even though the location/address has" and on the post-test, the explanation is "The functionF has pointer parameters so the values of pig and fox doesn't change but their addresses do." Similarly, on Q7, the pre-test explanation is "functionG passes the address of apple so in the function, the location containing the value is moved 3 places but the value stays the same..." and the post-test explanation echoes this with "The value of grape doesn't change since the value itself wasn't passed in functionG." With **B** coming into the intervention with working code not using pointers and **A** taking on the role of the HAG only providing confirmation feedback, this consistency makes sense. **B** also had a consistent PBV misconception with Q1 on the pre-test to post-test, saying on the pre-test "I can see in the main function, cat starts out to be 5 and dog 8. In functionA, a temporary variable is assigned the value of x which is 5, so temp is 5 (line 3). In line 4, x is assigned the value of y so x is 8. In line 5, y is assigned the temp variable which is 5. So y is 5 when the function returns, x (which is the cat) is now 8 and y (the dog) is 5." On the post-test, **B** echoes this, stating "In functionA, a temp variable is assigned the value of a (line 3), a to b (line 4) and b to the temp variable (line 5). The function then returns wolf which is b so its 6." **B** had one instance of reducing misconceptions where there is no evidence in the intervention to support the improvement on Q4. On the pre-test, **B** incorrectly said "In the functionD." On the post-test, **B** explained "The variable othervalue is assigned to the value of the a+b (line 3) in functionD so it's 2. value is still -5 since it remains in the main function." Video analysis does not support the intervention as the cause of the misconceptions reduced.

(331-141) Video Observations: In this case, **B** comes in with code that compiles and runs and

the pair goes through the HAG protocol as intended.

Language in Pre/Post-tests: This pair shows an interesting example for language changes. **A** exhibits improved language on Q5, relating to arrays. In the pre-test, **A** states "Since functionE is void, the array is not changed." In the post-test, however, **A** correctly explains "In functionE, a[0] is temporarily moved to num, a[0] becomes 12, and a[1] becomes -4." **B** on the other hand, exhibits consistent language from pre-test to post-test with regards to the misconceptions seen. On Q5, **B**'s pre-test response is "The function isn't changing anything outside it's scope." and the post-test response is "not a pointer to swap numbers." On Q8, **B** says "They aren't changing value in the function because it is outside of the scope" and on the post-test, the response is "there are no pointers." In both of these questions, **B** had issues with PBR and believed that arrays did not operate in a PBR manner. Considering this pair performed the HAG protocol as intended, this case seems to show **A** gaining benefits from a phenomenon outside of the scope of the experiment whereas **B** did not and performed as expected for a HAG participant (gaining no learning outcomes and keeping consistent misconceptions from pre-test to post-test).

(59-359) Video Observations: **B** comes into the protocol with code that compiles and runs. Since this is the case, **B** tests the code using **A** as a human autograder and it works as intended. Of note, when looking at the actual code in the video, it can be seen that this code was not implemented in a way that follows the instructions given on Day 1. **B** has included the print statements for the original and modified values in the flipSign function, so that there is not genuinely a flip that is happening, it is just temporarily happening during the flipSign function through a copy of the variables passed and then once it returns to the main function, the change goes away. This phenomena is interesting as it shows a result where confirmation feedback, commonly the feedback provided by autograders, can lead to no benefits if the code produces the correct output but is not implemented in the intended way. This result also supports previous findings where students are able to produce correct output of code without fully understanding the concepts[105]. This shortcut and not following instructions could have led the student to believe they had more of an understanding than they actually did and perhaps cemented their misconceptions.

Language in Pre/Post-tests: **B** in this case did not have any language changes from pre-test to post-test. The questions that were incorrectly answered on the post-test were not answered on the pre-test, so it cannot be claimed that there was less understanding. **A** actually exhibits worse

language in the post-test for Q3, saying "The final result is 5 because of the division inside of functionC." On the pre-test, **A** says "The final result is 0 is what will be printed because although the math was multiplied correctly in the function, there is already an int num declared w/ the value already set to 0." There is no evidence in the audio transcript of the intervention to support this acquisition of a misconception.

(212-183) Video Observations: This pair begins with **B** having code that does not compile and attempting to implement the program using PBR. In this interesting case, **B** ends up actually running the provided executable by accident as opposed to their own (which would not have worked correctly). Since the executable worked as intended, the pair begins going through the test cases.
Language in Pre/Post-tests: Both partners maintain their misconceptions from pre-test to post-test. **A** believes that arrays do not function in a PBR manner, saying on the pre-test for Q5 that "The return statement does verify what is returned thus cow does not change" and on the post-test "You didn't return anything in the function so it didn't change." **A** has a similar consistent language from pre-test to post-test for Q8, which is also incorrect. **B**'s language does not improve between pre-test or post-test either.

(94-312) Video Observations: In this pair, **B** begins the protocol with code that compiles and runs. The code worked as intended, but used PBV and twice called a flipSign function that returned a single value instead of using PBR. The pair goes through the intervention with **A** taking on the HAG as intended, making no relevant comments outside of providing inputs and outputs.
Language in Pre/Post-tests: **A** for the pre-test and post-test gave arithmetic rather than textual explanations. Still, **A** exhibited the same misconceptions in the post-test as the pre-test. **B** shows textual evidence of retaining the misconceptions from pre-test to post-test on questions surrounding arrays. For Q5, **B**'s explanation is "...functionE does not change the values of the elements in cow. Thus, after functionE returns, cow is unchanged.." On the post-test for Q5, **B** still holds this belief of arrays not being able to change values, stating "functionE is a void function. Since no pointers were used, the zebra array is unchaged." For Q8 on the pre-test (also arrays), **B** says "...cow[2] = 4, 7 and horse = 3 both are unchanged after functionH ends" and then on the post-test says "B/c of no pointers used in functionH, none of the changed values are permanent..." Working with the HAG was not able to help **B** address misconceptions that were held before the intervention.

(214-147) Video Observations: This pair begins with **B** having code that compiles and runs using pointers. **A** played the role of the HAG providing confirmation feedback as the pair went through the test cases as intended.

Language in Pre/Post-tests: For this pair, the pre-test and post-test for **A** were not available as they did not complete one of the tests. However, for **B**, there were no improvements in language. For most questions, there were no misconceptions exhibited by **B**, and for a few, there was not enough data (either no response or no explanation on a pre-test or post-test) to observe benefits.

(Unknown-39) Video Observations: This pair begins with **B** having code that compiles and runs using pointers but that has not implemented the looping functionality. After reading over the instructions, **B** tests the first test case with **A** providing the confirmation feedback that the code does not work (As there is no way to loop). **B** spends time trying to implement a do-while loop, eventually asking **A** "Do you know if this is the right way to do a do-while loop?" **B** then adds "Maybe I should just do a while loop." **B** then edits the code to add the looping functionality. After another round of testing, **B** realizes from the feedback that their code does not have the final line telling the user the number of operations performed. **B** goes back and adds this functionality. Following having a fully working program, the pair complete the intervention as intended with **A** providing confirmation feedback as the HAG.

Language in Pre/Post-tests: **B** used pointers to implement the solution and had no misconceptions on questions relating to pointers. **B** did have misconceptions about arrays, and since there was no feedback relating to this concept, **B** maintained these misconceptions from the pre-test to the post-test. On Q5, **B**'s explanation on the pre-test was "functionE switches the input array but does not return it so the array cow stays as 4, 7" and on the post-test, **B** stated "functionE didn't return anything but 0." This consistent belief that arrays do not have PBR functionality and particularly because the functions were void is echoed in the post-test, which is to be expected considering **B** only received confirmation feedback during the intervention.

A							B							
ID	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support	ID	Code working	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support
32		✓			✓		391		✓		✓	✓		
41							257	✓	✓					
113	✓		✓	✓			43		✓		✓	✓		
165							252	(By end)						
302	✓		✓	✓			153	(By end)		✓			✓	
338	✓		✓	✓			246	(By end)	✓	✓	✓	✓	✓	✓?
N/A	N/A	N/A	N/A	N/A	N/A	N/A	81							
N/A	N/A	N/A	N/A	N/A	N/A	N/A	131			✓			✓	
N/A	N/A	N/A	N/A	N/A	N/A	N/A	390	✓	Gained (Swap)	✓			✓	Swap Alg.

Figure 8.10: Overview of HAG Nots (Irrelevant Discussion)

8.2.3.2 HAG Nots (Irrelevant Discussion):

(Unknown-81) Video Observations: **B** begins having code that does not compile and spends the session trying to fix the code. It did not seem as if this was done collaboratively, with instead **B** debugging code and working through the issues.

Language in Pre/Post-tests: When actually looking at the pre- and post-test for this participant, it seems as if the "improvement" of scores might have been somewhat superficial, as there were questions that had numerous misconceptions on the pre-test that on the post-test just were not answered, therefore we could not accurately measure or assume misconceptions for the problems in the post-test.

(32-391) Video Observations: This pair had a technical issue and audio was lost for the video. During the video, it is seen that **B** begins with code that does not compile and spends the protocol time attempting to get the code to work. **B** attempted this using only one function, never declaring a flipSign function. **B** did not produce testable code, even by the end of the intervention.

Language in Pre/Post-tests: **B** exhibits the same misconceptions and language about pointers from pre-test to post-test. For Q6 in the pre-test, **B** says "The pointers didn't change the values of cow and duck." Similarly, for Q6 in the post-test, **B** remarks "The function doesn't affect the values of pig and fox." **B** retaining the same reasoning from pre-test to post-test is consistent with the expectations of the HAG group. **A** loses a PBV misconception on the post-test for Q1, saying "It does not change because it is a void function" whereas on the pre-test their explanation was "The function switches the values of the inputs." Without audio, there is no evidence to suggest this benefit came from the protocol or from testing, since they never had code to test.

(165-252) Video Observations: This pair begins with **B** having code that does not compile, initially having tried to implement the program using PBV and returning two integer values from the function. In the beginning, there are errors and **A** suggests moving the flipSign function to the top of the code as opposed to the bottom, as the compiler is not reading that there is a function before **B** is attempting to call it in main. The code then compiles but is not properly flipping the signs of the integers. **A** notices "it's not returning it right" and that there is an issue with attempting to return the two variables. **A** then suggests a workaround, where **B** could "call [the function] twice." There is still another issue with return values that **A** suggests fixing by assigning two new variables in main to equal flipSign. This solution allows **B** to get working code. From this point on, the pair completes the test cases with **A** acting as the HAG.

Language in Pre/Post-tests: Although the pair manages to successfully get the program working through discussion and some pair programming dynamics, there is no language change for the misconceptions exhibited in the pre-test.

(Unknown-131) Video Observations: In this case, the pair begins with **B** having code that does not compile but attempts to implement the solution using pointers. Time is spent to determine the appropriate algorithm to flip the signs of the integers. Eventually, **B** solves the problem by having conditionals for if the values are above or below zero, and a different algorithm to flip it depending on which value is input. From this point on, the pair completes the intervention with **A** taking on the intended role of HAG.

Language in Pre/Post-tests: **A** for this pair is unknown so their pre-test/post-test data are not available. **B** exhibits one improvement, on Q1, which relates to PBV. In the pre-test, **B** explains "the function switches the values of x and y,..." but for the post-test, changes the explanation to "No value is returned by the function." There is no evidence to support this language improvement in the protocol and **B** otherwise is consistent with the misconceptions and language observed.

(302-153) Video Observations: This pair begins with **B** having code that does not compile. The video was noticeably shorter than other videos, meaning maybe it was a pair who arrived to class late or had issues setting things up. **B** managed to get the program working using PBV and then the pair performed a few test cases with **A** acting as the HAG as intended until time ran out for

the intervention.

Language in Pre/Post-tests: **A** in this pair did not have any language improvements and retained the misconceptions from pre-test to post-test. **B** did seem to improve with the concept of pointers and them being PBR. On Q7, **B** said on the pre-test "The pointer is not changed but the static variable is," incorrectly believing that a PBV variable changes value and a PBR value does not. On the post-test for this same question, **B** now says "Only the pointer was changed." Video analysis does not support the intervention as the cause of the misconceptions reduced.

(113-43) Video Observations: This pair begins with **B** having code that does not compile. **B** had attempted to implement using a separate flipSign function, but could not get it figured out so then decided to use just the main function to accomplish the task. They spend some of the protocol time discussing the logistics of the protocol, not spending much time on the concepts of fixing the code. They eventually manage to get the code compiling and running, but there is an issue with the while loop. They test some cases and realize that it is going to be incorrect for all cases because of this issue. **B** then spends some time trying to fix the code and never gets it working as intended. A majority of time is spent doing the HAG protocol as expected, just with **A** telling **B** that their output is incorrect.

Language in Pre/Post-tests: Both partners retain a majority of their misconceptions or do not provide enough data to show improvement (no answers on pre-test or post-test for some questions). **A** retains misconceptions on Q4 relating to scope, stating on the pre-test "Both now hold the same value after the function is executed." and on the post-test "Value and other_value are both equal to the value of the function." Similarly, for Q5 that deals with arrays, **A** incorrectly reasons on the pre-test "The array is never changed through the function" and then on the post-test says "Because its the same as zebra." One change for **A** that is not supported by the protocol is the improvement on Q1 about PBV and the swap algorithm. On the pre-test, **A** remarks "The code puts the value of x assigned to dog's value before the value of y is changed." However, on the post-test, **A** says "The value of wolf doesn't change." No real reasoning is given behind why the value does not change on the post-test, so it is not necessarily a language improvement, but **A** does manage to correctly respond to Q1 on the post-test whereas on the pre-test the answer was incorrect.

(Unknown-390) Video Observations: The pair begins with **B** having code that compiles and runs using PBV and calling the flipSign function twice with an integer return value. They then

complete the protocol with **A** providing confirmation feedback as the HAG. The pair spends some time attempting to fix unnecessary things, such as when the inputs have leading "0's" and they want to be able to see the leading 0's in the output. These fixes were outside of the scope of the assignment and also do not relate to any of the misconceptions addressed in the study.

Language in Pre/Post-tests: **B** in this case had an instance of misconceptions being reduced and an instance of misconceptions being gained from pre-test to post-test, with neither supported by the feedback received. On Q1, **B** incorrectly reasoned in the pre-test that "temp is set to hold what value is in x or cat. Then x or cat is set to y or dog's value, this changes temp's value accordingly. So now temp holds y or dog's value. Then y or dog is set with the value in temp which has been since set to y or dog's value," not realizing that PBV only passes a copy so this swap does not stay after the function returns. On the post-test for Q1, **B** says "wolf will stay the same. the values will not be changed by the function." For Q5 about arrays, **B** correctly reasoned on the pre-test that "temp is set to 4, the first value in the array." On the post-test for this question, **B** says "any modifications to the array only occur in the function." What is interesting about these responses is that the logic and heuristic used stays consistent from pre-test to post-test. It seems as if in the pre-test, **B** believed that the swap algorithm could swap the actual values, which holds true for Q5 where arrays function like PBR, but not for Q1 where the variables are PBV. On the post-test, the heuristic has changed to not believing variables passed into a function can change values unless they are passed by pointers, which is true for Q1, but not for Q5, since arrays function in a similar manner to pointers with regards to allowing variables to change values. Video analysis does not support the intervention as the cause of the misconceptions reduced and gained.

(41-257) Video Observations: In this case, **B** comes into the protocol with code that compiles but does not originally work exactly as intended. **A** notices and comments how "oh... it didn't say 'you have performed 1 calculations"' **B** then goes to fix the print statement. After spending time discussing the print statements and loops, the pair eventually gets the code working as intended. At this point, they mostly behave in the intended HAG nature for the rest of the intervention.

Language in Pre/Post-tests: **B** improves in the post-test on Q1 and Q7 when it comes to PBV misconceptions. On the pre-test for Q1, **B** states "When the function is called temp is assigned 5 -cat then x is assigned y-value (8)." In the post-test, however, **B** says on Q1 "The function, functionA switches the a and b values but without pointers or a return type it won't change anything."

B similarly mentions pointers in the post-test of Q7, stating "grape is pointer and so its value is changed in main, while banana is not and thus remains 6 when called in main." Video analysis does not support the intervention as the cause of the misconceptions reduced.

(338-246) Video Observations: In this pair, **B** begins in with code that does not compile. Some time is spent to fix it, with **A** helping with the print and scan functions. After that part is fixed, they test it and realize that there is not the final line printing out the total number of operations performed and choose to go back and add this functionality. Eventually, the pair gets the code working using PBV and calling the flipSign function twice. After that point, they go through the HAG protocol as intended, with **A** providing confirmation feedback for the test cases.

Language in Pre/Post-tests: Both **A** and **B** exhibited the same misconceptions about arrays (Q5 and Q8) on the pre-test and the post-test. **B** on the pre-test for Q5 comments "functionE does not return anything to the main function, so cow is unaffected and remains the same as when initialized" and on the post-test for Q5 says "The array zebra is not changed by the function, since nothing is returned to the main function," almost identical language. A similar lack of improvement is seen on the language for Q8. **A** for Q8 on the pre-test explains "cow[0], cow[1] and horse are established in the main function and never changed" and on the post-test echoes this with "shark and bear are initialized as 6, 8 and 9 respectively and never changed." Similar to **B**, **A** has a lack of improvement on Q5 as well. **B** does seem to show improved certainty on questions relating to pointers, as in Q6 on the pre-test, **B** remarks "This is a complete guess. I forgot how pointers work in C. I think the * means what that points to changes." On the post-test, **B** says for Q6 "The function is affecting where the pointers are pointing, so the values change even though functionF returns nothing." There were no misconceptions for this question in either the pre-test or post-test, and no evidence in the protocol to support this improvement. Video analysis does not support the intervention as the cause of the misconceptions reduced. Based on the uncertainty exhibited in the pre-test and that this was done the first week into the semester, it is possible **B** felt uncomfortable with pointers after taking the pre-test and went home and reviewed them before coming to the second day of the intervention, although it cannot be verified as the improvement was outside of the scope of the experiment.

8.2.3.3 HAG Nots (Relevant Discussion):

A							B							
ID	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support	ID	Code working	Misconceptions maintained	Misconceptions reduced	Sustained language	Support	Improved Language	Support
101	None						45	(By end with pointers)	✓		✓	✓		
110							263	✓ (One function)	✓	✓			✓	
170	Gained		Worse	✓			201	✓+	None					
229	Gained		Worse	✓			239	✓ (One function)						
296	✓/Gained		✓/Worse	✓/✓?			67	(By end)		✓			✓	
305	✓		✓	✓			220		✓/Gained		✓/Worse	✓/✓		
306	✓		✓	✓			240		✓/Gained		✓/Worse	✓/✓		

Figure 8.11: Overview of HAG NOTs (Relevant Discussion)

(305-220) Video Observations: This pair begins with **B** having code that does not compile and using some syntax from a different programming language (C++). After getting the program working using just one function and no flipSign function, the pair begins the HAG tests as intended. At one point, there is discrepancy between **B**'s output and the expected output. **B** states "I never even changed the variables." "I just in the print statement said minus." **B** continues "Maybe because I never stored it as anything" "Cause I never actually changed the value." **A** replies "Yeah. You're just printing the value. You never stored it anywhere." The pair then continues and finishes the intervention as expected.

Language in Pre/Post-tests: **B** has a improvement and less misconceptions for Q5 about arrays after the intervention. On the pre-test, the response to Q5 is "array cow is unchanged because FunctionE returns nothing." On the post-test however, **B** now says "The two were flipped" and chooses the correct answer. Based on the protocol observation, it is possible that the pairs discussion about not changing the values and never storing them prompted **B** to realize that arrays do have the ability to store and change values, supporting this improvement. **A** does not exhibit any language improvement from pre-test to post-tests, and keeps the same misconceptions as expected from a HAG A participant.

(229-239) Video Observations: In this case, **B** begins with code that compiles and runs, although

there is not a separate flip function used as required in the instructions. After one test, it is noticed that **B** also has not included a print statement to tell the user the number of operations performed. **B** goes back and addresses this problem, and when trying to print count, includes an ampersand. **A** comments "You don't need the ampersand.." to which **B** remarks "You don't? It wouldn't work without it for me, so.." After attempting to compile and it not working, the pair think about the error and eventually **B** says "Let me just take out this right here. See if that fixes it." **B** then says "Oh yeah. It doesn't always need it. Ok, you're right," all referring to the ampersand. After successfully getting the code compiling, the pair goes through the test cases and **A** performs the HAG role as intended.

Language in Pre/Post-tests: **A** in this case performed worse on Q6 and Q7 after the HAG protocol. On Q6 for the pre-test, **A** explained "The values get passed through functionF." However, on the post-test for Q6, **A** says "The addresses of the variables are being changed and not the actual values." Based on the protocol, there is evidence that attempting to fix the code and seeing the error that arose when trying to put an ampersand before the count variable, **A** misconstrued this as passing variables with the ampersand symbol does not allow for the value to be changed.

(101-45) Video Observations: This pair begins with **B** having code that does not compile. **A** helps **B** get the code working through the use of pointers, reminding of syntax such as "make sure to dereference" and "then you can send the address." After attempting to compile, there is one more error, which **B** realizes how to fix, stating "Oh! Is it because I have this as an int and is it supposed to return something?" **A** tells **B** "You could either do void or you could do return 0." The pair soon gets the program working using pointers and an integer flipSign function that returns 0. After attempting to test a case, they realize the functionality to loop the program was not implemented. The pair fixes this issue after some time and working in a pair programming dynamic. They then complete the HAG test cases with **A** providing confirmation feedback as the HAG.

Language in Pre/Post-tests: Neither partner exhibited many misconceptions in the pre-test for this pair. The only observable one comes from **B** and is on Q1, relating to PBV. In the pre-test, **B**'s explanation is "... a temp variable is used to switch the values of x and y..." Similarly, in the post-test, **B** has the same misconception, stating "Wolf is set to equal the variable a in functionA." This pair did seem to benefit from working together and the social elements of discussing programming concepts to get the code working correctly, but since both partners had correct reason-

ing from the start, there was not enough room to see improved learning outcomes based on language.

(306-240) Video Observations: This pair begins with **B** having code that compiles but does not run properly and attempts to return two integers. They spend some time working on this issue, eventually realizing it was an issue with return values and implementing the function using PBV and calling the function twice, fixing some issues with the number of decimal places they were printing along the way. After the code works properly, the pair performs the HAG protocol as intended, with **A** providing the expected confirmation feedback.

Language in Pre/Post-tests: Both **A** and **B** exhibited the same misconceptions about arrays (Q5 and Q8) on the pre-test and the post-test. **A** on the pre-test for Q5 says "The array x[2] is not returned to the main function or assigned to cow." and on the post-test says "array a in functionE is not returned or assigned to array zebra," showing almost identical language. **A** exhibited a similar language having misconceptions for the other array question. **B** on Q8 for the pre-test says "The numbers assigned stay the same..." and on the post-test for the same question says "All values remain as set..." For Q5, **B** answered incorrectly but gave the reasoning "FunctionE changes 2 values of an array using a temp int." On the post-test, however, **B** reasons for Q5 that "The array is set to those values and never changed." Based on answering the question wrong in the pre-test and the reasoning given for Q8, it seems likely that **B** was exhibiting misconception language for Q5 and might have thought that the function changes the values only within the scope of that function, which would explain why **B** chose the response that the values were unchanged once the function ended. The pair showed language supporting that the confirmation feedback of autograders was not able to address their misconceptions.

(110-263) Video Observations: This pair begins with **B** having code that compiles and runs but uses only the main function and conditionals. They perform the HAG protocol as intended with **A** providing confirmation feedback. After going through multiple cases, **B** realizes that their count operation was not properly implemented. The pair discuss how to fix it, and **B** successfully gets this implemented. After this point, the pair finishes the test cases with **A** as the HAG.

Language in Pre/Post-tests: **A** has no noticeable change of misconceptions or language based on the unanswered questions in the pre-test and post-test. **B** shows improvement for both Q1 and Q6 in the post-test. For Q1, **B** originally says "In the function functionA, the variable temp is assigned the

value of the first variable, x, or in this case cat. cat is assigned the value of dog and dog is assigned the value of temp. cat is assigned to 8, the value of dog," walking through the swap algorithm and answering that the value of cat switches. In the post-test, **B** says "The function doesn't replace the value of wolf itself." For Q6, **B** on the pre-test first says "They remain the same because the function returns nothing," thinking that pointers do not operate in a PBR manner when the function is void. On the post-test, **B** explains "Pointers allow for variables to be changed directly." Video analysis does not support the intervention as the cause of the misconceptions reduced.

(170-201) Video Observations: This pair begins with **B** having code that compiles and runs implemented using pointers. **A** assumes the role of the HAG and provides confirmation feedback for **B** throughout the test cases. There were a few of the trickier test cases that do not seem to work for **B**'s program originally. When the group finished all of the cases, they went back to see what was wrong in the code. **B** shows the code to **A** and explains how the algorithm was very simple and so **B** would be unsure how to change anything. At this time, it is visible that **B** had an integer return type for the flipSign function although it used pointers and had that function return 0.

Language in Pre/Post-tests: **B** exhibited no language changes between pre-test and post-test and also had no notable misconceptions. **A** actually gained misconceptions about pointers on the post-test. For Q6 on the pre-test, **A**'s explanation was "Values 13 and 10 are put in functionF as x and y respectively. The lines then do simple math to change their values since x and y are pointers." On the post-test for the same question, **A** claimed "There is not a return in functionF." Considering **A** correctly answered Q7 (also relating to pointers) on the pre-test, this conceptual change seems to have come from the intervention. I would attribute it to **A** getting to look over **B**'s working code and noticing that **B** used an integer return type function as opposed to a void, causing **A** to develop the heuristic and misconception that pointers cannot change the values of variables if used in a void function.

(296-67) Video Observations: This pair begins with **B** having code that does not compile, but has attempted to implement the solution using pointers. They attempt to fix the code together, and after realizing it is not compiling correctly, **A** suggests "Maybe because um the pointer thing,.." **B** then decides to stop attempting PBR and try to implement the solution using PBV. Through this process, they are able to get code that compiles and prints the original values, but the modified (flipped) values are not printing correctly. The pair attempts to return two values from the flipSign

function, and also are fundamentally incorrect in thinking that PBV variables can operate in a PBR manner. After some time, **A** wonders "I forget if you can return two values." **B** responds with "well, even if it was, I feel like the first would be right but it's not." They continue to attempt to fix **B**'s code, without producing testable code by the end of the intervention.

Language in Pre/Post-tests: **A** maintains consistent language and PBV misconceptions for Q7, saying on the pre-test "Apple and orange are set equal to new values in the void functionG after they are sent from the main function" and on the post-test "The value of grape is sent as a pointer to the function and b is sent as a variable to the void function." **A** even calls out a difference in the post-test of grape being sent as a pointer and b being sent as a variable, but maintains the misconception that b, which is PBV, is manipulated as if it were PBR. On Q1, **A** gains a PBV misconception on the post-test, originally saying "cat = 5 is input into the void function from where it was defined in main function, cat remains equal to 5." but then on the post-test saying "In the functionA, the values of wolf and lion are switched." It is possible that working with **B** and never getting the correct answer caused a confusion about PBV, but there is no direct evidence in the transcript of the HAG protocol. **B** actually gains misconceptions relating to pointers that are supported by the protocol data. In the pre-test for Q6, **B** says "13 and 10 are put into the functionF and used in their respective equations 13 - 6 = 7 and 10 * 2 = 20 and since the values are pointers they point back to the address of cows and ducks." On the post-test, however, **B** explains "the values of pig and fox are arranged in the equations in functionF but there is no return statement so the original values are input." For Q7, **B** on the pre-test correctly responds "Orange stays the same as the value initialized in the main while apple changes because it is addressed to *x." Then on the post-test, **B** has the incorrect response that "the address stays the same for the pointer grape so its value does..." The discussion of pointers being what caused **B**'s program not to compile and then the pair managing to get compiling (albeit not working properly) code after switching from implementing with pointers seems to have ingrained in the mind of **B** that pointers do not allow values to be changed, only addresses. **B** also maintains misconceptions relating to return values, saying on the pre-test for Q4 "Since functionD returns a number after number is initially declared the original value disappears, leaving the resent [value] to be the same as othernumber." and then on the post-test for the same question saying "Value is changed in functionD from -5 to 2 and otherval is equal to the return value."

Table 8.19: Q6 Protocol Followers Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
True MBF (12)	9	0	100	**0.03/0.75**
HAGs A (7)	2	2	0	NA/0
HAGs B (9)	1	1	0	NA/0

Table 8.20: Q7 Protocol Followers Overview

Section	Tot. Pre Misc	Tot. Post Mis	% Changed	Paired T-test p-value/ Difference of Means
True MBF (12)	13	2	85	**0.03/0.92**
HAGs A (7)	4	6	-50	0.09/-0.29
HAGs B (9)	3	4	-33	0.17/-0.11

8.2.3.4 Other:

(54-264) Video Observations: This video lost sound a few minutes in, making analysis difficult. I can see from the video that **B** is attempting to fix code and it is possible that **A** is helping with this, but without the audio it is impossible to say for sure.

Language in Pre/Post-tests: With respect to language, **B** did not show any improvement from pre to post-test. **A** on the other hand mentions pointers in both the pre and post-test for Q6, but in the pre-test says that the values cannot change because they are passed by pointers whereas in the post-test mentions that passing by pointers allows for the change of the values in a void function.

8.2.4 Protocol Followers

After completing the thematic analysis, it was noted that the themes could be broken into subjects who adhered to the protocol and those who strayed away. For the MBF technique, the **Conversational (Semi On Prompt)** and the **Conversational (On Prompt**) groups were the two groups who did engage in the MBF technique as expected. For the HAG protocol, the **HAGs** were the group the adhered to the protocol and participated in providing/receiving confirmation feedback as expected. Additional quantitative analysis was done to observe the effects of the intervention when using the protocol follower groups for Q6 and Q7, the questions relating to PBR and PBV where the MBF prompts and intervention showed the most promising results.

When looking at the paired t-tests seen in Tables 8.19-8.20, it is clear that the **True MBFs**

Table 8.21: Q6 T-Tests and Difference of Means Between Groups for Protocol Followers

	True MBFs (12)
HAGs A (7)	**0.03/0.75**
HAGs B (9)	**0.03/0.75**

Table 8.22: Q7 T-Tests and Difference of Means Between Groups for Protocol Followers

	True MBFs (12)
HAGs A (7)	**0.009/1.20**
HAGs B (9)	**0.02/1.03**

significantly improved from pre-test to post-test on Q6 and Q7 and the HAGs did not. Using the t-tests to compare between groups as shown in Tables 8.21-8.22, the results show that the **True MBFs** significantly outperformed the HAGs regardless of whether the HAG participant was providing or receiving feedback on these questions.

Table 8.23: ANOVA for Q6 and Q7 for Protocol Follower Partner As

Question	**Condition (True MBF vs. True HAGs)**	**Test (Pre. vs. Post)**	**Test x Condition**
Q6	0.48	0.34	0.34
Q7	0.66	0.39	0.39

8.3 Discussion

Through this intervention, I have been able to show benefits that MBF provides, such as reducing the overall misconceptions significantly from pre-test to post-test, and specifically the benefits for questions relating to parameter passing with and without the use of pointers (PBV versus PBR). I have also found that the greater adherence to the protocol results in greater reduction in misconceptions for the MBF protocol, the hypothesized results. That is to say, the MBF groups that are conversational and on prompt result in more improved learning outcomes and the HAG groups that are true HAGs and do not engage in addition discussion result in more maintained outcomes as would be expected from a control group. This section describes the benefits seen in the MBF group, discusses how the emergent themes from the MBF and HAG groups affects the results, and gives concluding opinions on this study and future work that could be conducted.

Table 8.24: ANOVA for Q6 and Q7 for Protocol Follower Partner Bs

Question	Condition (True MBF vs. True HAGs)	Test (Pre. vs. Post)	Test x Condition
Q6	**0.04**	**0.007**	**0.007**
Q7	0.18	**0.007**	**0.007**

8.3.1 MBF vs. HAG Quantitative Improved Learning Outcomes

Quantitatively, I found significant improvement for the misconception based feedback (MBF) groups when comparing the number of post-test misconceptions to the number of pre-test misconceptions, and the results were stronger when comparing the "true MBF" (Conversational Semi On-Prompt and Conversational On-Prompt to the "true HAG" (HAGs). The concepts that were evaluated were PBV vs. PBR, Scope, Arrays, and Pointers. I found that MBF provided significantly greater improved learning outcomes than the human autograder (HAG) groups for questions and concepts relating to pointers and their use in PBR parameter passing. These quantitative learning outcomes are promising and suggest additional exploratory analysis for the MBF technique. Although only the PBR vs. PBV concepts showed statistically significant differences/improvements for the technique, the other concepts were either understood well (minimal misconceptions exhibited in the pre-test) or the EYR questions were not sufficiently challenging to have students exhibit multiple misconceptions on the questions. In practice, the design of the EYR pre and post-test questions, the lab assignments, and the prompts all rely on the instructor's knowledge of students' prior experience. This is and would be an iterative process that instructors would refine over time. Future iterations of the technique could work on question design for the EYR pre/post-tests that offer sufficient challenge to elicit multiple misconceptions on every question, which could result in statistical significance for concepts outside of parameter passing using PBR and PBV.

8.3.2 How MBF Categories Affect Results

From my final study, four categories of MBF emerged: Non-Conversational; Conversational (Off-Prompt); Conversational (Semi On-Prompt); and Conversational (On Prompt). Below, I discuss how these categories affected results and give suggestions for ideal pairings based on these results.

8.3.2.1 Non-Conversational:

This group comprised three pairs. For the most part, these groups saw the least amount of improvement in learning outcomes. Overall, there were no cases of misconceptions introduced between the partners, so although **B** was doing the majority of the talking and sometimes said incorrect things, there was no evidence of these incorrect beliefs being exhibited by **A** in the post-tests. All but one of the non-conversational pairs started with either code that did not compile or code that did not attempt to implement PBR. There were a few noticeable times that people in this group were able to gain benefits, either **B** from self explanation[30] or **A** from listening to **B** and picking up on the language and heuristics surrounding PBR, Pointers, and Scope. This group had multiple instances of **Support** within the intervention for the improved language, as seen in Figure 8.5.

These students did not follow the MBF protocol and thus did not incorporate the core feedback element of the technique as **A** is not providing feedback so much as being an interviewer for **B**. Any benefits from this group came from the prompts allowing **A** or **B** to realize some information they either had never learned (missing knowledge) or had forgotten (inert knowledge)[158]. To address this issue, instructors may wish to develop short videos walking through the MBF protocol to ensure students are not falling into this non-conversational group or hold training sessions where teaching assistants can help walk students through the protocol.

8.3.2.2 Conversational (Off-Prompt):

This group was comprised of only two pairs. Both of the pairs started out with code that did not work. The partners both were involved in speaking, but there was not much in depth discussion. A benefit noticed was that one of the pairs went into a pair programming dynamic and successfully were able to complete the code using arrays by the time the intervention ended. This group did not experience learning gains, but there were no misconceptions introduced as a result of the **Conversational (Off-Prompt)** theme of the MBF intervention.

The improvements in this group cannot be directly be attributed to the intervention as they mostly did not follow the given MBF prompts. Overall, this might be the least ideal pairing to have, as the students are essentially not participating in the technique and unable to address the misconceptions the prompts are developed to address. Instructors would want to find ways to avoid

this situation when using this technique, and although it may not always be feasible in larger classes to have knowledge of all of the students' comprehension of the concepts, a potential aid could be changing partners with frequency (i.e. every week or every time MBF is used for an assignment). Similar approaches of training sessions or videos could be adopted to solve this problem.

8.3.2.3 Conversational (Semi On-Prompt):

Only one pair that fell into this category. The pair begins with **B** having implemented the solution using PBV, but they then work together to attempt to use PBR while going through the prompts. The pair work together guided by the prompts when relevant to better understand the concepts while also using feedback from the compiler and internet to clear things up. This was evident in the improved learning outcomes exhibited by **B** in the post-test. The improvements here can directly trace back to the conversation and feedback that occurred during the MBF intervention, as described in the **Support** column of Figure 8.7. There was no introduction of misconceptions observable in this group. This group still does exhibit some pointer syntax issues while trying to work with the code, blindly applying operators (&, *) as seen in[43, 105].

Overall, this category had many positive outcomes from the intervention. Although this was not the intended way to utilize the MBF technique, having students guided by the prompts when relevant to them working on part of implementing a programming assignment as opposed to discussing them directly in the order they are given seems a promising way to benefit from this technique.

8.3.2.4 Conversational (On Prompt):

This category was comprised of five pairs. Of these five, there were various levels of completion of the assignment from not completed (3/5) to completed using PBV (1/5) to completed using pointers (1/5). Most of the pairs spent a significant amount of time discussing the Pointers cards, a side effect of them being the first set of cards that really delve into a topic that is commonly found to be difficult for students learning to program[105, 43, 78]. This group had the most relevant discussions, which seemed to help with improving their learning outcomes. Self-explanation appears to have helped many of the students in these pairs and also there were times when **A** was able to clear up misconceptions that **B** had about pointers when **B** had code that was not working. The prompts allowed groups to work as if they had hypothetically used pointers and some even decided

to attempt to implement the code using them as a result of the prompts. The discussion led by the prompts allowed for pairs to address most of the misconceptions for the topics that were discussed at length. Here it is also noticed that there is no introduction of misconceptions. The benefits gained from this group can also be traced back to the discussions that were happening during the intervention.

This group serves as a model for how the intervention was intended to work. It is promising that almost half of the pairs fell into either this category or the Conversational (Semi On-Prompt) category. This category represents a majority of the quantitative and qualitative improvement in learning outcomes observed to result from the technique, which helps to support its effectiveness.

8.3.3 How HAG Categories Affect Results

8.3.3.1 HAGs:

The pairs who performed the HAG protocol as expected had the hypothesized results for a control group, in that they maintained their misconceptions. In these pairs, there was noticeable evidence of the partners retaining their misconceptions and keeping consistent language for those misconceptions, as seen in the **Sustained Language** and **Support** columns of Figure 8.9. All of the groups showed some instance of the misconceptions and language staying consistent. In the four instances of change in misconception counts that were not hypothesized and there was not support in the video analysis of the protocol, one of the potential uncontrolled reasons mentioned at the end of section 8.2.2 could have occurred, such as one of the partners reviewing or gaining clarity from another source between the days of the intervention, or the act of participating in the experiment triggered the students to think about the concepts and perform better or worse on the post-test.

8.3.3.2 HAG Nots (Irrelevant Discussion):

The pairs who were in the HAG group but had discussions about programming concepts not relating to those addressed in the pre-/post-tests performed very similarly to the HAGs. All but one of these cases exhibited the hypothesized results on a majority of the questions with misconceptions, having consistent misconceptions and language from the pre-test to post-test, and **Support**, as seen in Figure 8.10 based on them being a control group. In the one case where the results were opposite to the hypothesis, the swap algorithm seemed to play a role in confusing the student, as also seen

with one of the students in the HAGs. This student had correctly answered Q1 on the pre-test and incorrectly answered Q5, but on the post-test had the reverse effect. As with the other student who showed this pattern, there was consistency in the belief that the swap algorithm would change the values regardless of PBV/PBR or that the algorithm would not be able to change the values. The other few instances in this HAG Nots group where there is improved language, there is no evidence in the intervention and one of the possible explanations for unsupported results could apply. In future work, students could directly be asked whether they engaged in additional review or practice since the pre-test. One case showed some evidence that there was review based on the language exhibited in the pre-test "This is a complete guess. I forgot how pointers work in C. I think the * means what that points to changes." and then the post-test "The function is affecting where the pointers are pointing, so the values change even though functionF returns nothing." The change in the certainty from pre-test to post-test suggests that some phenomenon occurred to make this student certain. This group generally did not show evidence of benefiting from the intervention, and instead, retained the misconceptions and language as hypothesized for only confirmation feedback or discussion irrelevant to the topics being tested.

8.3.3.3 HAG Nots (Relevant Discussion):

This group showed the most dynamic range of results. Of the seven pairs who were categorized this way, there were three cases where the relevant discussion led to evidence of gained misconceptions and language that did not improve, one case where it led to evidence of fewer misconceptions and improved language, two cases where there were no pre-test misconceptions on the topic discussed during the intervention, and one case where there is an improvement with no supporting evidence from the intervention. These discussions made it difficult to clearly evaluate the actual differences between autograders and MBF, which are the two scenarios intended to be compared. This group shows how adding the human element of the autograder can result in unexpected differences. The group not only left the script of the expected HAG protocol, but had conversations that seem to have affected their responses on the post-tests. As seen in the results, this can either work out to the benefit or detriment to learning. It does not provide as much consistent benefit as the structured and guided nature of the MBF technique.

8.4 Conclusions

None of the MBF categories caused students to have negative learning outcomes that were supported by the qualitative analysis of the video transcripts. Even in the least ideal scenarios (Non-Conversational and Conversational (Off-Prompt)), there were no instances of observable misconceptions that were introduced based on the conversations that occurred during the intervention. The two ideal groups (Conversational (Semi On-Prompt) and Conversational (On Prompt) exhibited significant improvements both quantitatively and qualitatively. Students were able to reduce their misconceptions in the post-test from the pre-test and the language used to explain the answers on the post-test improved from the pre-test, which can be in many cases directly traced back to the conversations students engaged in during the intervention. One suggestion would be to not have pairs where neither partner had much understanding of the concepts coming into the activity. The one group that I observed this with had very little discussion and no learning gains were particularly possible. Any other pairing seemed to be able to provide benefits, regardless of which partner had a better grasp on concepts coming into the intervention. Instructors could feasibly randomly assign pairs. Another suggestion would be to encourage students to implement the solution to an assignment in different ways and then to assign partners to students who used different implementation methods. The ideal situation allows for students to be able to use the prompts to actually engage in discussion, so instructors may wish to redesign the prompts or how the technique is administered to foster an environment more conducive to discussion. This could be made possible by having both partners display their code instead of one, and allowing them to both respond to the provided prompts.

8.4.1 MBF vs HAG Improved Language

When looking at the language between the two groups, the MBF group exhibited more improved language that was supported by their intervention than the HAG groups. Even in instances where the HAG groups did have conversations relating to programming concepts (HAG NOTs), they either discussed things irrelevant to the concepts addressed during the pre-tests and post-tests (HAG NOTs (Irrelevant Discussion)) or discussed concepts addressed during these tests but those discussion led to as many cases of gained misconceptions as reduced (HAG NOTs (Relevant Discussion)). The MBF groups, particularly the conversational semi on-prompt and on prompt, had

relevant discussions guided and prompted by the MBF technique, and then used what was learned from these discussions to exhibit improved language on the post-tests, usually also being accompanied with a reduction in misconceptions. MBF participants also had no instances of language getting worse. The HAGs group in general maintained the same language from the pre-test to the post-test. The few instances where the language improved on the post-test were mostly not supported by the conversations in the interventions, and there were several instances where participants had worse language on the post-test that was supported by the pairs having unprompted discussions during the intervention. Overall, as hypothesized, the MBF group, which was intended to have discussion based on misconceptions, exhibited a better grasp of the concepts as evidenced by the language used compared to the HAG group.

8.4.2 Limitations

This work was conducted with one course at a specific type of institution, student population, concepts, and misconceptions searched for, so claims cannot be generalized for other environments. Replicating this study changing factors such as the demographics of the students, type of institutions, concepts/misconceptions observed, and the potential to completely randomize both the treatment groups and ensure the students are randomly assigned with similar levels would be beneficial to increase the generalizability of results. Although this work is motivated by the desire to help bridge the gap for underrepresented groups (URGs) in CS, this study was limited in not having a proper population to test the benefits for URGs. Proper usage of this methodology would allow practitioners who want to replicate the study to develop the materials based on the population they are studying. Replicating the study at minority serving institutions (MSIs), such as HBCUs and Hispanic Serving Institutions (HSIs), would provide data to aid in generalizing the benefits to URGs. Another limitation is the human element of the autograders. Although this methodology choice was made to control for the instruction style, as introducing a third variable of feedback being provided by a human for one condition and a machine for the other would have confounded the study design, having a human simulate an autograder did result in instances of discussions not intended for the protocol. Replicating the study with an autograder software used as the control would address this limitation and provide a direct comparison to what is a more realistic alternative. In the future, a study could be designed to test feedback style across instruction style (machine designed to provide MBF vs. machine autograder giving confirmation feedback). Another study could examine the ben-

efits of the prompts themselves versus the interactive nature by having a computer provide just the prompts to respond to in one group and in another having a computer provide the prompts and also interact/have discussion with the student. This would help to tease apart some of the differences of the benefits, such as how important is the social constructivism aspect of having a peer to discuss the prompts vs. being able to discuss and go through the process with a machine. The design of the experiment being carried out over two days is another limitation to consider. Students had the opportunity to review material between the first (pre-test) and second (post-test) day of the experiment. However, students were not instructed to review between the days and all students had the same opportunity to review. A final limitation to mention exists in the methodology. Some students did not answer all of the questions in either the pre-test or post-test or did not provide explanations. Although I attempted to correct for this in my closed coding, there was still some noise in calculating the change of misconceptions that existed from the pre-test to the post-test. Replicating the study with a larger study group would help to provide more data and validate the results of thee study.

8.4.3 Future Work

This work leads to promising scenarios for future work. One noteworthy observation was how many pairs in the MBF intervention spent much more time on the Pointers cards than the Array cards. While this did lead to significant results in the Pointers category, it would be worthwhile to examine Arrays or other concepts and how successful the technique is at addressing misconceptions in those categories. Based on analyzing the audio and video of the interventions, I believe that the reason pairs spent more time on Pointers is that the prompts were given in a way so that the Pointers card pile was before the Arrays pile. I believe that the Pointers prompts in many cases provided enough fodder for discussion that pairs did not have enough time to adequately discuss the Arrays cards. To rectify this, the index cards could be placed on the desk beforehand so that they are presented in no set order. Another option is developing a virtual version of this technique. This version would allow students to choose the topics/prompts they want to go through and instructors could assign certain topics to students to focus on particular concepts/misconceptions. To introduce a more engaging element to this modification of the technique, it could be developed as a *Choose Your Own Adventure* game. A virtual version would also make it easier to have the prompts not feel as if they are interview questions, as both students could display the prompts as well as their code.

This would help address the issue of students falling into the Non-Conversational category. However, the virtual version might reduce the level of human interaction, which was shown to benefit not only the students who followed the MBF protocol, but also the students who did not adhere to the HAG protocol and had discussions.

Another consideration would be to experiment with interleaving the MBF technique with an autograder. There were noticeable benefits that came from students who decided to test aspects of their code while working. This relates back to previous work that shows the benefits of interacting with the compiler[105, 80]. In this version of the technique, students would be encouraged to test out via the use of an autograder their various ways of implementing the solution to a programming problem while discussing the concepts. The autograder could help correct some of the uncertainty, as having the confirmation feedback supporting their self explanations and MBF dialogue would strengthen their conceptions. For this particular lab exercise, this would allow students to see if the signs had been flipped for various test cases and also work on the concepts of scope while ensuring that their counting operation functioned properly. The autograder would need to prepared in a way to ensure that correct solutions could not exist without a proper understanding. A consideration in a similar vein is to integrate the use of MBF prompts into an autograder/compiler. This would lead to a more refined and peer-based version of Gusakuma's misconception-based compiler feedback[80]. In a similar augmented version, MBF could be tweaked to provide immediate feedback while students performed pair programming[214]. As Epstein found in[59], immediate feedback shows learning benefits for students, and a study comparing MBF giving immediate feedback to MBF as evaluated in my dissertation would help tease apart which benefits can be attributed to when the feedback is received versus benefits that are tied to students being able to construct and take ownership of their own code. Of interest to researchers wishing to replicate this MBF study would be using actual autograders as opposed to the human autograder approach used in this study. This would allow the experiment to eliminate the variable nature of humans being able to have conversations that have been shown in this study to potentially be helpful or detrimental to their learning.

Future work also exists outside of the scope of this one intervention I have developed and evaluated. Looking back to my formative studies, there are multiple suggestions that would be worth examining in the future. It was noted in my coding in the wild study in Chapter 5 that students exhibited difficulties with arrays and using them to change values in different functions, at times just deciding to avoid the use of arrays totally. These difficulties persisted in my final evaluative

study, and with many students not discussing the Arrays prompts as often, the misconceptions were not able to be addressed. Instructors may wish to ensure that students gain experience with non-standard cases with arrays and the idea of pass by reference semantics, allowing them to work through the difficulties and ensuring that they understand the concepts. A particular case that would be valuable to address is when the swap algorithm is present. I observed multiple instances in my final study of students being confused by the use of swap algorithm, either thinking that it always swaps the actual values or never does. Since arrays behave in a PBR manner, this adds an additional layer of complexity which potentially causes cognitive overload.

Another common difficulty seen in the coding in the wild study was the blind application of syntax when it came to pointers. This was echoed in my final study as there still seemed to be instances of students not understanding when to use a "&" versus "*." To address this issue, instructors may wish to provide additional opportunities for students to work with pointers and allow the students to discuss, explain, and at times justify why certain syntax is used in certain situations. This opportunity to have conversations and address the misconceptions early on has proven to offer the benefit of addressing misconceptions through the MBF group. Although the developed prompts did attempt to address dereferencing (*) vs. passing an address (&), a simple but potentially effective change would be to explicitly use the symbols while having the students discuss the use of each when first learning pointers.

A general recommendation for these misconceptions and difficulties students are facing while learning CS is to ensure instructors are aware of them so that they can be properly addressed. As mentioned in related worked, misconceptions are persistent and resistant to change, meaning that just telling students they are wrong as a way to address the misconceptions will not be effective. Students need to be provided the opportunities to form their own correct schemas through experiences. This constructivist approach, whether through active learning, discovery learning, inquiry-based learning, or some other form of constructivism-based pedagogical approach, is the way that has shown to effectively address misconceptions.

8.4.4 Contributions and Final Thoughts

This dissertation presents one quasi-experimental study of how MBF can provide learning outcome benefits for students who have misconceptions, particularly related to PBR and pointers. From a more generalized idea, instructors could develop prompts using a different lab assignment that

focuses on different concepts. For example, a programming assignment to implement a bank that provided loans to people based on various interest rates could be the problem. This assignment could be broken down into the types of bank accounts, the algorithms/functions for how to calculate interest, and keeping track of the total calculations performed. Following a similar structure as seen in 7.1, instructors could develop questions that first focus on ensuring students understand the syntax of the functions and concepts necessary to implement the solution. The questions would then allow the students to analyze and apply those concepts through discussion, and at the end, offering them the opportunity to evaluate the decisions they either have made or would have made.

This dissertation presents the motivation, background, and evaluation to support a new active learning technique to use in CS classrooms. I have established a "recipe"/methodology to develop and implement an active learning technique centered around misconceptions in CS topics. Through designing and evaluating my MBF technique, I provide a pedagogical technique to address the rising enrollments of CS courses without sacrificing conceptual learning benefits. An active learning technique allows for everyone to benefit, but can particularly benefit underrepresented groups (URGs), who can be hurt most by approaches that focus on limiting enrollment or on quick feedback rather than quality. As seen in the work by [82] and [126], active learning has been shown to provide benefits to all, but have particular benefits to URGs. MBF offers students a way to learn and gain conceptual benefit from peers, similar to peer instruction[44], one of the active learning techniques with demonstrated benefits for URGs. Although my evaluation of MBF did not have the demographic variety to look at effects on URGs and so direct benefit cannot be claimed, it is likely that techniques such as MBF would help to bridge the gap of URGs underperforming in CS courses. Addressing these difficulties that are common in introductory CS courses can help students complete the first two years of programs, at which point there is a better chance of them staying in the major[142]. I hope to continue work to provide all students, and hopefully especially URGs, tools to grasp the content and make them feel comfortable to avoid the identity issues associated with navigating a field that doesn't have much representation. After having authored with two of my committee members a book chapter on how active learning has been and can be applied in a CS context, the development of this MBF technique offers a novel approach to not only CS active learning, but STEM learning in general. This reusable development of active learning technique can be applied to STEM courses that have some form of activity associated with them. For chemistry, physics, or biology, it could be a laboratory assignment similar to a programming lab. For

mathematics, it could be a set of problems that relate to certain concepts and misconceptions. The benefits of developing a structured set of prompts based on the misconceptions and how knowledge of novices learn combined with allowing students to have a discourse to address those misconceptions should theoretically apply across disciplines. These are issues that students continue to face while navigating through learning how to program and STEM courses. I intend to continue researching to bridge the gap that exists not just in CS, but in STEM fields in general, with a desire to intersect discipline-based education research with science policy. This dissertation is my first step on the way towards this goal.

Appendices

Appendix A Conceptual Assessment

Name:

Start Time _____ **End Time _____**

Instructions: For the following questions, you will be asked to either choose between two answer choices or provide your own answer. Following each answer, you should describe your thought process in arriving at your answer.

You may assume that all code snippets compile. Do not go back after answering a question.

An example is provided below.

Example: Consider the following code snippet below:

```
1.  int main(){
2.    int number, sign;
3.    printf("Please type in a number: ");
4.    scanf ("%d", &number);
5.    if (number < 0)
6.      sign = -1;
7.    else
8.      sign = 1;
9.    printf ("Sign = %i\n", sign);
10.   return 0;
11. }
```

Assume the user inputs the value "3." What is the result of the execution of this code?

a. "Sign is -1"
b. "Sign is 1"

Explain your reasoning below:

In this example, I can see that there are "if" and "else" statements in lines 5 and 7. The "number" variable that is given by the user is checked to see if it is less than 0 in line 5. Since we are assuming the number is 3, it is not less than 0 and does not meet the "if" condition. So we look at the else condition and assign the value 1 to the variable sign, which is then printed.

1. Consider the following code snippet:

```
1. int x;
2. x = 5 % 3;
3. printf ("x is %d", x);
```

What is the result of the execution of this code?

Explain your reasoning below:

2. Consider the following code snippet below:

```
1. int i, j, k, m;
2. i = 10;
3. j = 10;
4. k = i++;
5. m = ++j;
```

What are the values of the variables **k** and **m** after this code executes?

k:

m:

Explain your reasoning below:

3. Consider the following snippet of code:

```
1.  int Scores[3];
2.
3.  Scores[0] = 1;
4.  Scores[1] = 2;
5.  Scores[2] = 3;
6.
7.  if (Scores[1 + 1] == 2 )
8.    printf ("This answer.");
9.  else
10.   printf ("That answer.");
```

What is the result of the execution of this code?

a. This answer.
b. That answer.

Explain your reasoning below:

4. Consider the following snippet of code:

```
1. int x;
2. x = 6;
3.
4. if (x = 7)
5.     printf (“first choice”);
6. else
7.     printf (“second choice”);
```

What is the result of the execution of this code?

Explain your reasoning below:

5. Consider the following code snippet:

```
1. int x, y;
2. x = 13;
3. y = 21;
4.
5. if (20 > y - x)
6.    printf ("Thing One");
7. else
8.    printf ("Thing Two");
```

What is the result of the execution of this code?

a. Thing One
b. Thing Two

Explain your reasoning below:

6. Consider the following code snippet below:

```
1.  int cow, moon, star;
2.  cow = 4;
3.  moon = 1;
4.
5.  if ((cow == 1 + 5) && (7 > moon)) {
6.     star =0;
7.  }
8.
9.  else {
10.    star = 10;
11. }
```

What is the value of the variable **star** after this code executes?

a. 0
b. 10

Explain your reasoning below:

7. Consider the following code segment:

```
1.  int a, b, c, d;
2.  a = 5;
3.  b = 3;
4.  c = 6;
5.  d = 2;
6.  if (a < d)
7.      b = c;
8.  if (d< b)
9.      c = a;
10. d = b;
```

What are the values of b, c, and d after this code executes?

b =
c =
d =

Explain your reasoning below:

8. Consider the following code

```
1.  int fish, dog, i;
2.  fish = 4;
3.  dog = 2;
4.  for (i = 0; i < 3; i++) {
5.    fish++;
6.    dog--;
7.    if (dog < 0) {
8.       int temp;
9.       temp = dog;
10.      dog = fish;
11.      fish = temp;
12.   }
13. }
```

At the end of this code, what are the values of the variables **fish** and **dog?**

fish:
dog:

Explain your reasoning below:

9. Consider the following function definition:

int measure (double x, double y);

What type of value does this function return?

a. double
b. integer

Explain your reasoning below:

10. Consider the following function definition:

int measure (char x, char y);

What type of variable does this function take in as a parameter?

a. character
b. integer

Explain your reasoning below:

11. Consider the following code segment:

```
1.  void swap (int x, int y){
2.    int temp;
3.    temp = x;
4.    x = y;
5.    y = temp;
6.    return;
7.  }
8.
9.  int main( ) {
10.  int cat, dog;
11.  cat=5;
12.  dog =8;
13.  swap (cat, dog);
14.  return 0;
15. }
```

What is the value of variable **cat** after the **swap** function returns?

a. 5
b. 8

Explain your reasoning below:

12. Consider the following code segment:

```
1.  int subtract (int x, int y){
2.    int answer;
3.    answer = x -y;
4.    return answer;
5.  }
6.
7.  int main( ) {
8.    int cat, dog, solution1, solution2;
9.    cat=5;
10.   dog =8;
11.   solution1 = subtract (cat, dog);
12.   solution2 = subtract (dog, cat);
13.   printf ("Solution1 is %d\n", solution1);
14.   printf ("Solution2 is %d", solution2);
15.   return 0;
16. }
```

What is the result of the execution of this code?

Explain your reasoning below:

13. Consider the following code snippet:

```
1.  int volume (int height, int width, int length) {
2.     int solution;
3.     solution = height * width * length;
4.     return solution;
5.  }
6.
7.
8.  int main() {
9.  int x, y, z, vol;
10. x = 3;
11. y = 2;
12. z = 4;
13. vol = volume (z, x, y);
14. return 0;
15. }
```

In this example, what is the value of the variable **width** in the **volume** function?

width:

Explain your reasoning below:

14. Considering the following code snippet below:

```
1. void findArea(int length, int width) {
2. int area;
3. area = length * width;
4. return;
5. }
6.
7. int main () {
8. int x, y, area;
9. x = 4;
10. y = 8;
11. area = 0;
12. findArea(x, y);
13. printf("The area of the shape is %d", area);
14. return 0;
15. }
```

What is the result of the execution of this code?

Explain your reasoning below:

15. Consider the following code snippet below:

```
1. void findArea(int length, int width) {
2. int area;
3. area = length * width;
4. printf (“The area of the shape is %d\n”, area);
5. return;
6. }
7.
8. int main () {
9. int x, y;
10. x = 3;
11. y = 5;
12. findArea(4, 6);
13. findArea(x, y);
14. return 0;
15. }
```

What is the result of the execution of this code?

Explain your reasoning below:

16. Consider the following code snippet below:

```
1.  int sum (int num1, int num2) {
2.    int answer;
3.    answer = num2 + num1;
4.    return answer;
5.  }
6.
7.  int main () {
8.    int solution;
9.    solution = sum (9, 4);
10.   solution = sum (5, 3);
11.   printf("The solution is %d", solution);
12.   return 0;
13. }
```

What is the result of the execution of this code?

Explain your reasoning below:

17. Consider the following code segment below:

```
1.  void subtract (int x, int y){
2.    int answer;
3.    answer = x -y ;
4.    return;
5.  }
6.
7.  int main( ) {
8.    int answer;
9.    answer = 7;
10.  subtract (8, 5);
11.  printf ("Answer is %d", answer);
12.  return 0;
13. }
```

What is the result of the execution of this code?

a. Answer is 3
b. Answer is 7

Explain your reasoning below:

18. Consider the following code segment below:

```
1.  int answer = 7;
2.
3.  int subtract(int x, int y){
4.    int answer;
5.    answer = x -y  ;
6.    return answer;
7.  }
8.
9.  int main( ) {
10.  int cat, dog, solution;
11.  cat = 5;
12.  dog = 8;
13.  solution = subtract (cat, dog);
14.  printf ("Answer is %d", answer);
15.  return 0;
16. }
```

What is the result of the execution of this code?

a. Answer is -3
b. Answer is 7

Explain your reasoning below:

19. Consider the following code snippet below:

```
1.  int maximum (int test[3]) {
2.     int max, i;
3.     max = test[0];
4.     for (i = 1; i < 3; i++) {
5.        if (test[i] > max)
6.           max = test[i];
7.        }
8.     return max;
9.  }
10.
11. int main () {
12.    int answer;
13.    int num[3] = {13, 31, 19};
14.    answer = maximum (num);
15.    printf ("The maximum value is %d", answer);
16.    return 0;
17. }
```

What is the result of the execution of this code?

Explain your reasoning below:

20. Consider the following code snippet below:

```
1.  int add (int x, int y){
2.    int answer;
3.    answer = x +y  ;
4.    return answer;
5.  }
6.
7.  int main( ) {
8.    int cat, dog, solution;
9.    cat = 5;
10.   dog = 8;
11.   add(cat, dog);
12.   printf("Solution is %d", solution);
13.   return 0;
14. }
```

What is the result of the execution of this code?

Explain your reasoning below:

21. Consider the following code segment:

```
1.  int add(int a, int b) {
2.    int sum, answer;
3.    answer = 7;
4.    sum = a + b;
5.    return sum;
6.  }
7.
8.  int main( ) {
9.    int x, y, answer;
10.  x = 4;
11.  y = 9;
12.  answer = add (x, y);
13.  return 0;
14. }
```

What values does the variable named **answer** take on over the course of this program's execution? Use program lines to make it clear what part you're discussing. Explain your reasoning behind each value:

Appendix B Live Coding Instruction Page

Instructions: Run the executable using the command:

- ./splay

Try out each of the operations to get a feel for the program/s functionality.

Now recreate the functionality of the executable, using separate functions to accomplish each operation (subtraction, addition, flip sign).

Remember to think aloud and explain what you are doing while working through the task.

Below is an example of a code snippet that you can use to get you started on the framework:

```
1. #include <stdio.h>

2. int main(void){

3. printf("Hello, world\n");

4. /*
5. We get you started with a simple call to an addition function
6. */

7. //printf("%d + %d  = %d\n", x, y, add(x,y));

8. }
```

NOTE: Your print statements should be in the **main** function and not in the separate functions. Your call to the addition function does not have to look exactly like line 7.

Appendix C Explain Your Reasoning Pre-Test

Name:

Start Time _____ **End Time _____**

Instructions: For the following questions, you will be asked to either choose between two possible answers or to provide your own answer. Following each question, you should describe your thought process in arriving at your answer.

You may assume that all code snippets compile. Please answer the questions in order. Do not return to earlier questions.

An example is provided below.

Example: Consider the following code snippet below:

```
1.  int main(){
2.    int number;
3.    printf("Please type in a number: ");
4.    scanf ("%d", &number);
5.    if (number % 2 == 0)
6.      printf("This is even");
7.    else
8.      printf("This is odd");
9.    return 0;
10. }
```

Assume the user inputs the value "3." What is the result of the execution of this code?

a. "This is even"
b. "This is odd"

Explain your reasoning below:

In this example, I can see that there are "if" and "else" statements in lines 5 and 7. The expression number % 2 is checked to see if it is equal to 0 in line 5. Since we are assuming the number is 3, number % 2 is not equal to 0 and does not meet the "if" condition. So we look at the else condition and see that what should be printed is "This is odd."

1. Consider the following code segment:

```
1.  void functionA (int x, int y){
2.    int temp;
3.      temp = x;
4.      x = y;
5.      y = temp;
6.      return;
7.  }
8.
9.  int main( ) {
10.   int cat, dog;
11.     cat=5;
12.     dog =8;
13.     functionA (cat, dog);
14.     return 0;
15. }
```

What is the value of variable **cat** after the **functionA** function returns?

a. 5
b. 8

Explain your reasoning below:

2. Consider the following code segment:

```
1.  int functionB (int x, int y){
2.      x = x / y;
3.      return x;
4.  }
5.
6.  int main( ) {
7.    int moon, cat, number;
8.      moon = 12;
9.      cat = 4;
10.     number = functionB (moon, cat);
11.     printf ("The value of number is %d", number);
12.     return 0;
13. }
```

What will be printed from line 11?

a. The value of number is 12
b. The value of number is 3

Explain your reasoning below:

3. Consider the following code segment:

```
1.  void functionC (int a, int b) {
2.    int num;
3.      num = a * b;
4.      return;
5.  }
6.
7.  int main () {
8.    int x, y, num;
9.      x = 4;
10.     y = 8;
11.     num = 0;
12.     functionC (x, y);
13.     printf("The final result is %d", num);
14.     return 0;
15. }
```

What will be printed from line 13?

Explain your reasoning below:

4. Consider the following code segment:

```
1.  int functionD (int x, int y){
2.    int number;
3.       number = x - y;
4.       return number;
5.  }
6.
7.  int main( ) {
8.    int number, other_number;
9.       number = 7;
10.      other_number = functionD (8, 5);
11.      printf  ("Number is %d.  Other number is %d.", number, other_number);
12.      return 0;
13. }
```

What will be printed from line 11?

Number is _____. Other number is _____.

Explain your reasoning below:

5. Consider the following code segment:

```
1.  void functionE (int x[2]){
2.    int temp;
3.       temp = x[0];
4.       x[0] = x[1];
5.       x[1] = temp;
6.       return;
7.  }
8.
9.  int main( ) {
10.   int cow[2] = {4, 7};
11.      functionE (cow);
12.      return 0;
13. }
```

What is the value of array **cow** after **functionE** returns?

a. {4, 7}
b. {7, 4}

Explain your reasoning below:

6. Consider the following code segment:

```
1.  void functionF (int* x, int* y){
2.    *x = *x - 6;
3.    *y = *y * 2;
4.  }
5.
6.  int main( ) {
7.    int cow = 13;
8.    int duck = 10;
9.       functionF (&cow, &duck);
10.     return 0;
11. }
```

What are the values of variables **cow** and **duck** after **functionF** returns?

cow:

duck:

Explain your reasoning below:

7. Consider the following code segment:

```
1.  void functionG (int* x, int y){
2.    *x = *x + y;
3.     y = y / y;
4.  }
5.
6.  int main( ) {
7.    int apple = 7;
8.    int orange = 3;
9.       functionG (&apple, orange);
         return 0;
10. }
```

What are the values of variables **apple** and **orange** after **functionG** returns?

apple:

orange:

Explain your reasoning below:

8. Consider the following code segment:

```
1.  int functionH (int x[2], int y){
2.      y = x[0];
3.      x[1] = y * 2;
4.      return x[0];
5.  }
6.
7.  int main( ) {
8.    int cow[2] = {4, 7};
9.    int horse = 3;
10.  int num;
11.     num = functionH (cow, horse);
        return 0;
12. }
```

What is the values of the variables **cow[0], cow[1], horse,** and **num** after **funcionH** returns?

cow[0]:

cow[1]:

horse:

num:

Explain your reasoning below:

Appendix D Explain Your Reasoning Post-Test

Name:

Partner A or B:

Start Time _____ **End Time _____**

Instructions: For the following questions, you will be asked to either choose between two answer choices or provide your own answer. Following each answer, you should describe your thought process in arriving at your answer.

You may assume that all code snippets compile. Do not go back after answering a question.

An example is provided below.

Example: Consider the following code snippet below:

```
a.      int main(){
b.        int number;
c.        printf("Please type in a number: ");
d.        scanf ("%d", &number);
e.        if (number % 2 == 0)
f.          printf("This is even");
g.        else
h.          printf("This is odd");
i.        return 0;
j.      }
```

Assume the user inputs the value "3." What is the result of the execution of this code?

a. "This is even"
b. "This is odd"

Explain your reasoning below:

In this example, I can see that there are "if" and "else" statements in lines 5 and 7. The expression number % 2 is checked to see if it is equal to 0 in line 5. Since we are assuming the number is 3, number % 2 is not equal to 0 and does not meet the "if" condition. So we look at the else condition and see that what should be printed is "This is odd."

1. Consider the following code segment:

```
1.  void functionA (int a, int b){
2.    int num;
3.      num = a;
4.      a = b;
5.      b = num;
6.      return;
7.  }
8.
9.  int main( ) {
10.   int lion, wolf;
11.     lion = -6;
12.     wolf = 2;
13.     functionA (lion, wolf);
14.     return 0;
15. }
```

What is the value of variable **wolf** after the **functionA** function returns?

a. -6
b. 2

Explain your reasoning below:

2. Consider the following code segment:

```
1.  int functionB (int a, int b){
2.     a = a * b;
3.     return a;
4.  }
5.
6.  int main( ) {
7.     int sky, cloud, value;
8.        sky = -10;
9.        cloud = 3;
10.       value = functionB (sky, cloud);
11.       printf ("The value of value is %d", value);
12.       return 0;
13. }
```

What will be printed on line 11?

a. The value of value is -30
b. The value of value is -10

Explain your reasoning below:

3. Consider the following code segment:

```
1.  void functionC (int x, int y) {
2.    int num;
3.      num = x / y;
4.      return;
5.  }
6.
7.  int main () {
8.    int a, b, val;
9.      a = -10;
10.     b = -2;
11.     val = 0;
12.     functionC (a, b);
13.     printf("The final result is %d", val);
14.     return 0;
15. }
```

What will be printed from line 13?

Explain your reasoning below:

4. Consider the following code segment:

```
1.  int functionD (int a, int b){
2.     int value;
3.        value = a + b;
4.        return value;
5.  }
6.
7.  int main( ) {
8.     int value, other_value;
9.        value = -5;
10.       other_value = functionD (-4, 6);
11.       printf ("Value is %d.  Other value is %d", value, other_value);
12.  return 0;
13. }
```

What will be printed from line 11?

Value is _____. Other value is _____.

Explain your reasoning below:

5. Consider the following code segment:

```
1.  void functionE (int a[2]){
2.    int num;
3.      num = a[0];
4.      a[0] = a[1];
5.      a[1] = num;
6.  }
7.
8.  int main( ) {
9.    int zebra[2] = {-4, 12};
10.     functionE (zebra);
        return 0;
11. }
```

What is the value of array **cow** after the **FunctionE** returns?

a. {12, -4}
b. {-4, 12}

Explain your reasoning below:

6. Consider the following code segment:

```
1.  void functionF (int* a, int* b){
2.    *a = *a + 4;
3.    *b = *b / 2;
4.  }
5.
6.  int main( ) {
7.    int pig = -3;
8.    int fox = 8;
9.    functionF (&pig, &fox);
      return 0;
10. }
```

What are the values of variables **pig** and **fox** after the **functionF** returns?

pig:

fox:

Explain your reasoning below:

7. Consider the following code segment:

```
1.  void functionG (int* a, int b){
2.    *a = *a - b;
3.     b = b * b;
4.  }
5.
6.  int main( ) {
7.     int grape = -4;
8.     int banana = 6;
9.        functionG (&grape, banana);
          return 0;
10. }
```

What are the values of variables **grape** and **banana** after **functionG** returns?

grape:

banana:

Explain your reasoning below:

8. Consider the following code segment:

```
1.  int functionH (int a[2], int b){
2.     b = a[0];
3.     a[1] = b / 2;
4.     return a[0];
5.  }
6.
7.  int main( ) {
8.     int shark[2] = {6, -8};
9.     int bear = 9;
10.    int val;
11.       val = method (shark, bear);
          return 0;
12. }
```

What is the values of the variables **shark[0], shark[1], bear,** and **val** after **functionH** returns?

shark[0]:

shark[1]:

bear:

val:

Explain your reasoning below:

Appendix E Final Study Coding Task Instructions

Instructions:

1. Log in using your Clemson credentials.
2. Have your RecordMyDesktop set up for you by one of the TAs or researchers.
3. Open a Terminal window (ask for help if needed).
4. Type **`mkdir CPSC1020`** and press Enter
5. Type **`cd CPSC1020`** and press Enter
6. Type **`mkdir Lab1`** and press Enter
7. Type **`cp ~cazembk/public_html/flip .`** and press Enter.
8. Run the executable by typing **`./flip`** and press Enter.

Try out the operation a few times to get a feel for the program's functionality.

Now recreate the functionality of the executable, using a SEPARATE **flipSign** function to accomplish the flipping operation. You may use any editor you are comfortable using.

Your program should be named **userID_flip.c.**

For example, my program would be named **cazembk_flip.c**

Below is an example of a code snippet that you can use to get you started:

```
#include <stdio.h>

int main(void) {

   printf("Hello, world\n");

  return 0;

}
```

NOTE: Your print statements should be in the **main** function and not in the **flipSign** function.

Submitting your work:

You will submit your video file of your recorded screen.

To submit your file, you should:

1. Go to a Terminal window (ask for help if needed).
2. Type **`cd`** and press Enter.
3. Type **`ls *.ogv`** and press Enter.
4. You should see a file named **`out.ogv`** or similar.
5. Type **`mv out.ogv yourusername.ogv`** and press Enter.
    a. For example, I would type **`mv out.ogv cazembk.ogv`** and press Enter.
6. Type **`ls *.ogv`** and press Enter to confirm that you have renamed the file.
7. Type **`cp yourusername.ogv /group/vidcap/fall2019`** and press Enter to submit the file.
    a. For example, I would type

    `cp cazembk.ogv /group/vidcap/fall2019` and press Enter.

Appendix F Intervention Instructions

Misconception-Based Feedback Instructions

- A *TA or researcher* will set up **Partner B's** machine to capture video & audio.
- **Partner B** should wear the microphone headset as demonstrated by the researcher.
- **Partner A** should hold the card deck.
- **Partner B** should bring up the code solution from the last Thursday's lab.
- Work ONLY on **Partner B's** machine (Partner A should not even log in).
- Discuss how **Partner B** implemented the solution.
- **Partner A** should use the cards, in the order provided, to guide the discussion.
- **Partner B** may choose to modify their code as the discussion progresses.
- The interaction should be conversational, with **Partner A and Partner B** both sharing their knowledge, opinions, strategies, etc. Feel free to use the index cards for notes or sketches to aid your discussion.
- If a question on a particular card does not elicit responses from either Partner, **Partner A** should move on to the next question.

Submitting your work:

You will submit your video file of your recorded screen that includes the audio of your discussions.

To submit your file, you should:

1. Have a researcher or TA help you stop and save your video file
2. Go to a Terminal window (ask for help if needed).
3. Type **`cd`** and press Enter.
4. Type **`ls *.ogv`** and press Enter.
5. You should see a file named **`out.ogv`** or similar.
6. Type **`mv out.ogv PartnerBusername-PartnerAusername.ogv`** and press Enter.
    a. For example, if I was Partner B and Dr. Kraemer was Partner A, I would type

    `mv out.ogv cazembk-etkraem.ogv` and press Enter.

6. Type **`ls *.ogv`** and press Enter to confirm that you have renamed the file.
7. Type **`cp RenamedFileName.ogv /group/vidcap/fall2019`** and press Enter to submit the file.
    a. For example, I would type

    `cp cazembk-etkraem.ogv /group/vidcap/fall2019` and press Enter.

Human Autograder Instructions

- A *TA or researcher* will set up **Partner B's** machine to capture video & audio.
- **Partner B** should wear the microphone headset as demonstrated by the researcher.
- **Partner A** should hold the card deck.
- **Partner B** should bring up the code solution from the last Thursday's lab. Be sure that the code compiles and that it executes, takes in inputs, and produces outputs.
- If **Partner B** does not have code that is ready to test, **Partner B** should comment out lines to simplify the program to a point where it is testable.
- Work ONLY on **Partner B's** machine (Partner A should not even log in).
- **Partner A** will read the following test cases, in order, and Partner B will enter the inputs provided.
- **Partner A** will display what is the back of the card, compare with Partner B's screen output and say either "Correct" or "Incorrect."
- **Partner B** may choose to modify their code based on the test results.

Submitting your work:

You will submit your video file of your recorded screen that includes the audio of your discussions.

To submit your file, you should:

1. Have a researcher or TA help you stop and save your video file
2. Go to a Terminal window (ask for help if needed).
3. Type **`cd`** and press Enter.
4. Type **`ls *.ogv`** and press Enter.
5. You should see a file named **`out.ogv`** or similar.
6. Type **`mv out.ogv PartnerBusername-PartnerAusername.ogv`** and press Enter.
    a. For example, if I was Partner B and Dr. Kraemer was Partner A, I would type

    `mv out.ogv cazembk-etkraem.ogv` and press Enter.
7. Type **`ls *.ogv`** and press Enter to confirm that you have renamed the file.
8. Type **`cp RenamedFileName.ogv /group/vidcap/fall2019`** and press Enter to submit the file.
    a. For example, I would type

    `cp cazembk-etkraem.ogv /group/vidcap/fall2019` and press Enter.

Appendix G Intervention Prompts

Partner A should hold the card deck.

Partner B should bring up the code solution from the last Thursday's lab.

Discuss how Partner B implemented the solution.

Partner A should use the cards, in the order provided, to guide the discussion.

Partner B may choose to modify their code as the discussion progresses.

The interaction should be conversational, with Partner A and Partner B both sharing their knowledge, opinions, strategies, etc. Feel free to use the index cards for notes or sketches to aid your discussion.

If a question on a particular card does not elicit responses from either Partner, Partner A should move on to the next question.

Overall Program Cards

1. What is the overall flow of control of the program?
2. Can you walk through the code/function?
    a. Where do you create variables?
    b. What are their types?

Variables/Values

3. How do you get values from the user? Where/How do you display those values on the screen?
4. Where do your variables change values?

Function Call/Parameter Passing/Return Values

5. How do you send values (pass parameters) to the flip function?
6. How do you flip those values?
7. What is the return type of your flip function?
8. Do you return values from the flip function? How? Where are they stored?
9. How are the flipped values accessed from the main function?
10. Reflect on your solution – did you use pass-by-value? Pass-by-reference? Something else?
    a. If pass-by-reference, did you use arrays or pointers?
        i. **If arrays, Partner A should read from Arrays card deck now**
        ii. **If pointers, Partner A should read from Pointers card deck now**
        iii. **If neither and time remains, Partners A and B should look through the Arrays and Pointers cards and discuss how the solution might have been implemented using the questions as a guide.**

Pointers Cards:

1. Where are the pointers created?
2. What are they pointing to? How did the pointer take on that value?
3. Where are pointers dereferenced?

4. Do you pass an address or a value as a parameter in the function call? What's the difference? What is the type of the parameter/s passed into your function?
5. How are the addresses and their values accessed and/or changed inside the function?
6. Are any values in the main function modified as a result of the flip function's execution? How does that happen?
7. How are pointers and references related? Different?
8. Do you create any reference variables?
9. Where are the reference variables created?
10. Where are they assigned a value? What do they refer to?
11. Do you use reference variables in passing parameters to a function?
12. How are the reference variables used inside the flip function?
13. Are any values returned from the function by using a reference variable?
14. **If Partner B used pointers and time remains, Partners A and B should go through the Arrays cards and discuss how the solution might have been implemented using the questions as a guide. If you have gone through both Pointers and Arrays, move on to Operation Count and Code Cleanup Cards.**

Arrays Cards:

1. Where is the array declared?
2. What values are stored in the array? Where in the code does that happen?
3. How many elements can the array contain?
4. What are the indices of the array? What values are at each index location?
5. Do you pass an array element as a parameter? If so, how?
6. Do you pass the array as a parameter? If so, how?
7. Are the values in the array different after the function call returns? How does that happen?
8. **If Partner B used arrays and time remains, Partners A and B should go through the Pointers cards and discuss how the solution might have been implemented using the questions as a guide. If you have gone through both Pointers and Arrays, move on to Operation Count and Code Cleanup Cards.**

Operation Count

1. How do you know when to stop running the program?
2. How do you keep track of how many times you performed the flip operation?
3. How and when do you display the count of how many times you ran the flip operation?

Code Cleanup

1. Do you use all of the variables you've declared?
2. Do you ever declare two variables of the same name in different functions?
    a. If yes, how does assigning a value to one affect the other?
    b. If no, how would assigning a value to one affect the other if you did declare two variables of the same name in different functions?
3. Do you ever use global variables?
    a. Where were they initialized?
    b. Where were they used?
    c. How did their values change?

Test Inputs (HAGs)

Partner A should hold the card deck.

Partner B should bring up the code solution from the last Thursday's lab. Be sure that the code compiles and that it executes, takes in inputs, and produces outputs.

If Partner B does not have code that is ready to test, Partner B should comment out lines to simplify the program to a point where it is testable.

Partner A will read the following test cases, in order, and Partner B will enter the inputs provided.

Partner A will display what is the back of the card, compare with Partner B's screen output and say either "Correct" or "Incorrect."

Test Case 1:

Please enter the first integer: **3**

Please enter the second integer: **7**

Would you like to continue? Press 1 for yes, any other number for no: **4**

Output Case 1:

The original values are 3 and 7

The modified values are –3 and –7

Thanks! You have performed 1 calculations.

Test Case 2:

Please enter the first integer: **-385**

Please enter the second integer: **12**

Would you like to continue? Press 1 for yes, any other number for no: **-56**

Output Case 2:

The original values are -385 and 12

The modified values are 385 and –12

Thanks! You have performed 1 calculations.

Test Case 3:

Please enter the first integer: **0**

Please enter the second integer: **-8462**

Would you like to continue? Press 1 for yes, any other number for no: **473**

Output Case 3:

The original values are 0 and -8462

The modified values are 0 and 8462

Thanks! You have performed 1 calculations.

Test Case 4:

Please enter the first integer: **-0**

Please enter the second integer: **0**

Would you like to continue? Press 1 for yes, any other number for no: 0

Output Case 4:

The original values are 0 and 0

The modified values are 0 and 0

Thanks! You have performed 1 calculations.

Test Case 5:

Please enter the first integer: **-592**

Please enter the second integer: **70000000000**

Would you like to continue? Press 1 for yes, any other number for no: **797**

Output Case 5:

The original values are -592 and 1280523264

The modified values are 592 and -128052364

Thanks! You have performed 1 calculations.

Test Case 6:

Please enter the first integer: **-93**

Please enter the second integer: **-17**

Would you like to continue? Press 1 for yes, any other number for no: **4**

Output Case 6:

The original values are -93 and -17

The modified values are 93 and 17

Thanks! You have performed 1 calculations.

Test Case 7:

Please enter the first integer: **99**

Please enter the second integer: **56871**

Would you like to continue? Press 1 for yes, any other number for no: **-3**

Output Case 7:

The original values are 99 and 56871

The modified values are -99 and -56871

Thanks! You have performed 1 calculations.

Test Case 8:

Please enter the first integer: **0**

Please enter the second integer: **0**

Would you like to continue? Press 1 for yes, any other number for no: **60000000000**

Output Case 8:

The original values are 0 and 0

The modified values are 0 and 0

Thanks! You have performed 1 calculations.

Test Case 9:

Please enter the first integer: **-8000000000000**

Please enter the second integer: **-72942**

Would you like to continue? Press 1 for yes, any other number for no: **98765**

Output Case 9:

The original values are -1524072448 and -72942

The modified values are 1524072448 and 72942

Thanks! You have performed 1 calculations.

Test Case 10:

Please enter the first integer: **0123**

Please enter the second integer: **-7654**

Would you like to continue? Press 1 for yes, any other number for no: **-2**

Output Case 10:

The original values are 123 and -7654

The modified values are -123 and 7654

Thanks! You have performed 1 calculations.

Test Case 11:

Please enter the first integer: **0099**

Please enter the second integer: **-964**

Would you like to continue? Press 1 for yes, any other number for no: **-35**

Output Case 11:

The original values are 99 and -964

The modified values are - 99and 964

Thanks! You have performed 1 calculations.

Test Case 12:

Please enter the first integer: **2147483647**

Please enter the second integer: **214783648**

Would you like to continue? Press 1 for yes, any other number for no: **-322573**

Output Case 12:

The original values are 2147483647 and -2147483648

The modified values are -2147483647 and -2147483648

Thanks! You have performed 1 calculations.

Test Case 13:

Please enter the first integer: **214783648**

Please enter the second integer: **213783649**

Would you like to continue? Press 1 for yes, any other number for no: **90210**

Output Case 13:

The original values are -214783648 and -214783647

The modified values are -214783648 and 214783647

Thanks! You have performed 1 calculations.

Test Case 14:

Please enter the first integer: **-2147483647**

Please enter the second integer: **-2147483648**

Would you like to continue? Press 1 for yes, any other number for no: **-21**

Output Case 14:

The original values are -2147483647 and -2147483648

The modified values are 2147483647 and -2147483648

Thanks! You have performed 1 calculations.

Test Case 15:

Please enter the first integer: **-2147483648**

Please enter the second integer: **-2147483649**

Would you like to continue? Press 1 for yes, any other number for no: **3**

Output Case 15:

The original values are -2147483648 and 2147483647

The modified values are -2147483648 and -2147483647

Thanks! You have performed 1 calculations.

Test Case 16a:

Please enter the first integer: **23**

Please enter the second integer: **-908**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 16a:

The original values are 23 and -908

The modified values are –23 and 908

Test Case 16b:

Please enter the first integer: **748**

Please enter the second integer: **58**

Would you like to continue? Press 1 for yes, any other number for no: **0**

Output Case 16b:

The original values are 748 and 58

The modified values are –748 and –58

Thanks! You have performed 2 calculations.

Test Case 17a:

Please enter the first integer: **673**

Please enter the second integer: **-718**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 17a:

The original values are 673 and -718

The modified values are –673 and 718

Test Case 17b:

Please enter the first integer: -**74**

Please enter the second integer: **7429**

Would you like to continue? Press 1 for yes, any other number for no: 985

Output Case 17b:

The original values are -74 and 7429

The modified values are 74 and –7429

Thanks! You have performed 2 calculations.

Test Case 18a:

Please enter the first integer: -**7259**

Please enter the second integer: **-0081943**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 18a:

The original values are -7259 and -81943

The modified values are 7259 and 81943

Test Case 18b:

Please enter the first integer: **480**

Please enter the second integer: **0**

Would you like to continue? Press 1 for yes, any other number for no: **0683**

Output Case 18b:

The original values are 480 and 0

The modified values are –480 and 0

Thanks! You have performed 2 calculations.

Test Case 19a:

Please enter the first integer: **301**

Please enter the second integer: **-412**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 19a:

The original values are 301 and -412

The modified values are –301 and 412

Test Case 19b:

Please enter the first integer: **5894**

Please enter the second integer: **010101**

Would you like to continue? Press 1 for yes, any other number for no: **7555**

Output Case 19b:

The original values are 5894 and 10101

The modified values are –5894 and –10101

Thanks! You have performed 2 calculations.

Test Case 20a:

Please enter the first integer: **-8943**

Please enter the second integer: **-4591324**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 20a:

The original values are -8943 and -4591324

The modified values are –8943 and 4591324

Test Case 20b:

Please enter the first integer: **04**

Please enter the second integer: -**09**

Would you like to continue? Press 1 for yes, any other number for no: **-076**

Output Case 20b:

The original values are 4 and -9

The modified values are –4 and 9

Thanks! You have performed 2 calculations.

Test Case 21a:

Please enter the first integer: **-911**

Please enter the second integer: **411**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 21a:

The original values are -911 and 411

The modified values are 911 and -411

Test Case 21b:

Please enter the first integer: -**00411**

Please enter the second integer: **00911**

Would you like to continue? Press 1 for yes, any other number for no: **-611**

Output Case 21b:

The original values are -411 and 911

The modified values are 411 and -911

Thanks! You have performed 2 calculations.

Test Case 22a:

Please enter the first integer: **-1000000000000 (12)**

Please enter the second integer: **600000000000 (11)**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 22a:

The original values are 727379968 and -1295421440

The modified values are –727379968 and 1295421440

Test Case 22b:

Please enter the first integer: **-000**

Please enter the second integer: **987**

Would you like to continue? Press 1 for yes, any other number for no: **-076**

Output Case 22b:

The original values are 0 and 987

The modified values are 0 and -987

Thanks! You have performed 2 calculations.

Test Case 23a:

Please enter the first integer: **-4**

Please enter the second integer: **13**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 23a:

The original values are -4 and 13

The modified values are 4 and 13

Input Case 23b:

Please enter the first integer: **1098765**

Please enter the second integer: **-5**

Would you like to continue? Press 1 for yes, any other number for no: **23**

Output Case 23b:

The original values are 1098765 and -5

The modified values are -1098765 and 5

Thanks! You have performed 2 calculations.

Test Case 24a:

Please enter the first integer: -000

Please enter the second integer: **00300**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 24a:

The original values are 0 and 300

The modified values are 0 and -300

Test Case 24b:

Please enter the first integer: **235**

Please enter the second integer: **-135711**

Would you like to continue? Press 1 for yes, any other number for no: **23**

Output Case 24b:

The original values are 235 and -135711

The modified values are -235 and 135711

Thanks! You have performed 2 calculations.

Test Case 25a:

Please enter the first integer: **1889**

Please enter the second integer: **1867**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 25a:

The original values are 1889 and 1867

The modified values are -1889 and -1867

Test Case 25b:

Please enter the first integer: **-00100**

Please enter the second integer: **-00200**

Would you like to continue? Press 1 for yes, any other number for no: **32143**

Output Case 25b:

The original values are -100 and -200

The modified values are 100 and 200

Thanks! You have performed 2 calculations.

Test Case 26a:

Please enter the first integer: **3183**

Please enter the second integer: **-1241**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 26a:

The original values are 3183 and -1241

The modified values are 3183 and 1241

Test Case 26b:

Please enter the first integer: **0**

Please enter the second integer: **003600**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 26b:

The original values are 0 and 3600

The modified values are 0 and -3600

Test Case 26c:

Please enter the first integer: **-00**

Please enter the second integer: **-365**

Would you like to continue? Press 1 for yes, any other number for no: **143**

Output Case 26c:

The original values are 0 and -365

The modified values are 0 and 365

Thanks! You have performed 3 calculations.

Test Case 27a:

Please enter the first integer: 42

Please enter the second integer: -7777

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 27a:

The original values are 42 and -7777

The modified values are -42 and 7777

Test Case 27b:

Please enter the first integer: 3

Please enter the second integer: 2

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 27b:

The original values are 3 and 2

The modified values are -3 and -2

Test Case 27:

Please enter the first integer: **-3**

Please enter the second integer: **-2**

Would you like to continue? Press 1 for yes, any other number for no: **-1**

Output Case 27c:

The original values are -3 and -2

The modified values are 3 and 2

Thanks! You have performed 3 calculations.

Test Case 28a:

Please enter the first integer: **-0000**

Please enter the second integer: **-1111**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 28a:

The original values are 0 and -1111

The modified values are 0 and 1111

Test Case 28b:

Please enter the first integer: -**0834**

Please enter the second integer: **8675309**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 28b:

The original values are -834 and 8675309

The modified values are 834 and -8675309

Test Case 28c:

Please enter the first integer: **-623**

Please enter the second integer: **-525600**

Would you like to continue? Press 1 for yes, any other number for no: **2133**

Output Case 28c:

The original values are -623 and -525600

The modified values are 623 and 525600

Thanks! You have performed 3 calculations.

Test Case 29a:

Please enter the first integer: -711411911

Please enter the second integer: 119114117

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 29a:

The original values are -711411911 and 119114117

The modified values are 711411911 and -119114117

Test Case 29b:

Please enter the first integer: **04**

Please enter the second integer: **30**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 29b:

The original values are 4 and 30

The modified values are -4 and -30

Test Case 29c:

Please enter the first integer: **-314**

Please enter the second integer: **-500**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 29c:

The original values are -314 and -500

The modified values are 314 and 500

Test Case 29d:

Please enter the first integer: **1**

Please enter the second integer: **1**

Would you like to continue? Press 1 for yes, any other number for no: **23**

Output Case 29d:

The original values are 1 and 1

The modified values are -1 and -1

Thanks! You have performed 4 calculations.

Test Case 30a:

Please enter the first integer: **-9**

Please enter the second integer: **8**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 30a:

The original values are -9 and 8

The modified values are 9 and -8

Test Case 30b:

Please enter the first integer: **404**

Please enter the second integer: **-075**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 30b:

The original values are 404 and -75

The modified values are -404 and 75

Test Case 30c:

Please enter the first integer: **44**

Please enter the second integer: **-44**

Would you like to continue? Press 1 for yes, any other number for no: **1**

Output Case 30c:

The original values are 44 and -44

The modified values are -44 and 44

Test Case 30d:

Please enter the first integer: **199**

Please enter the second integer: **6**

Would you like to continue? Press 1 for yes, any other number for no: **2**

Output Case 30d:

The original values are 199 and 6

The modified values are -199 and -6

Thanks! You have performed 4 calculations.

Overall Program Cards	McCracken	SOLO	Bloom's
1. What is the overall flow of control of the program?	Abstract problem	Relational	Evaluate
2. Can you walk through the code/function?	Subproblems	MS	Analyze
a. Where do you create variables?	Subsolutions	US	Understand
b. What are their types?	Abstract problem	US	Understand
3. How do you get values from the user?	Subproblems	MS	Apply
Where/How do you display those values on the screen?	Subsolutions	MS	Understand
4. Where do your variables change values?	Subsolutions	US	Understand
5. How do you send values (pass parameters) to the flip function?	Subproblems	MS	Apply
6. How do you flip those values?	Subproblems	MS	Apply
7. What is the return type of your flip function?	Subproblems	US	Understand
8. Do you return values from the flip function? How? Where are they stored?	Abstract problem/Evaluate	Relational	Evaluate
9. How are the flipped values accessed from the main function?	Subsolutions	MS	Apply
10. Reflect on your solution – did you use pass-by-value? Pass-by-reference? Something else?	Abstract problem/Evaluate	Relational	Evaluate
a. If pass-by-reference, did you use arrays or pointers?	Subsolutions	US	Remember
Pointers Cards:			
1. Where are the pointers created?	Subsolutions	US	Understand
2. What are they pointing to? How did the pointer take on that value?	Subsolutions	MS	Apply
3. Where are pointers dereferenced?	Subsolutions	US	Understand
4. Do you pass an address or a value as a parameter in the function call? What's the difference? What is the type of the parameter/s passed into your function?	Abstract problem	Relational	Evaluate
5. How are the addresses and their values accessed and/or changed inside the function?	Abstract problem	MS	Analyze
6. Are any values in the main function modified as a result of the flip function's execution? How does that happen?	Evaluate	MS	Evaluate
7. How are pointers and references related? Different?	Abstract problem	MS	Analyze
8. Do you create any reference variables?	Subsolutions	US	Understand
9. Where are the reference variables created?	Subsolutions	US	Understand
10. Where are they assigned a value? What do they refer to?	Subsolutions	US	Understand
11. Do you use reference variables in passing parameters to a function?	Subsolutions	MS	Apply
12. How are the reference variables used inside the flip function?	Subsolutions	MS	Apply
13. Are any values returned from the function by using a reference variable?	Evaluate	MS	Evaluate
Arrays Cards:			
1. Where is the array declared?	Subsolutions	US	Understand
2. What values are stored in the array? Where in the code does that happen?	Subsolutions	MS	Apply
3. How many elements can the array contain?	Subsolutions	US	Understand
4. What are the indices of the array? What values are at each index location?	Subsolutions	MS	Apply
5. Do you pass an array element as a parameter? If so, how?	Evaluate	MS	Evaluate
6. Do you pass the array as a parameter? If so, how?	Evaluate	MS	Evaluate
7. Are the values in the array different after the function call returns? How does that happen?	Evaluate	MS	Evaluate
Operation Count			
1. How do you know when to stop running the program?	Abstract problem	Relational	Analyze
2. How do you keep track of how many times you performed the flip operation?	Subsolution	MS	Apply
3. How and when do you display the count of how many times you ran the flip operation?	Subsolution	MS	Apply
Code Cleanup			
1. Do you use all of the variables you've declared?	Subsolution	US	Understand
2. Do you ever declare two variables of the same name in different functions?	Evaluate	Relational	Analyze
a. If yes, how does assigning a value to one affect the other?	Evaluate	Relational	Analyze
b. If no, how would assigning a value to one affect the other if you did declare two variables of the same name in different functions?	Evaluate	Relational	Analyze
3. Do you ever use global variables?	Subsolution	US	Understand
a. Where were they initialized?	Subsolution	US	Understand
b. Where were they used?	Subsolution	US	Understand
c. How did their values change?	Subsolution	MS	Apply

Figure 12: All Questions

Bibliography

[1] Lorin W Anderson, David R Krathwohl, Peter W Airasian, Kathleen A Cruikshank, Richard E Mayer, Paul R Pintrich, James Raths, and Merlin C Wittrock. A taxonomy for learning, teaching, and assessing: A revision of bloom's taxonomy of educational objectives, abridged edition. *White Plains, NY: Longman*, 2001.

[2] Evelyne Andreewsky and Danièle Bourcier. Abduction in language interpretation and law making. *Kybernetes*, 29(7/8):836–845, 2000.

[3] Computing Research Association et al. Generation cs: Computer science undergraduate enrollments surge since 2006, 2017.

[4] Richard C Atkinson and Richard M Shiffrin. Human memory: A proposed system and its control processes. 1968.

[5] David Paul Ausubel, Joseph Donald Novak, Helen Hanesian, et al. Educational psychology: A cognitive view. 1968.

[6] Alan Baddeley. Working memory (vol. 11), 1986.

[7] Albert Bandura and Richard H Walters. *Social learning theory*, volume 1. Prentice-hall Englewood Cliffs, NJ, 1977.

[8] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, September 1983.

[9] John B Biggs and Kevin F Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, 2014.

[10] Paul Black. Formative assessment: Raising standards inside the classroom. *School Science Review*, 80(291):39–46, 1998.

[11] Glenn D Blank, Sally Hiestand, and Fang Wei. Overcoming misconceptions about computer science with multimedia. In *Proceedings of 35th SIGCSE Technical Symposium on Computer science Education*. Citeseer, 2004.

[12] Donald A Bligh. *What's the Use of Lectures?* Intellect books, 1998.

[13] David Boud and Elizabeth Molloy. *Feedback in higher and professional education: understanding it and doing it well*. Routledge, 2013.

[14] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Threshold concepts in computer science: do they exist and are they useful? *ACM SIGCSE Bulletin*, 39(1):504–508, 2007.

[15] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[16] Bo Brinkman and Amanda Diekman. Applying the communal goal congruity perspective to enhance diversity and inclusion in undergraduate computing degrees. In *Proceedings of the 47th ACM technical symposium on computing science education*, pages 102–107. ACM, 2016.

[17] Peter C Brown, Henry L Roediger III, and Mark A McDaniel. *Make it stick*. Harvard University Press, 2014.

[18] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[19] Jerome S Bruner. The act of discovery. *Harvard educational review*, 31:21–32, 1961.

[20] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. The collaborative nature of pair programming. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 53–64. Springer, 2006.

[21] Duane Buck and David J Stucki. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. 2000.

[22] Duane Buck and David J Stucki. Jkarelrobot: a case study in supporting levels of cognitive development in the computer science curriculum. *ACM SIGCSE Bulletin*, 33(1):16–20, 2001.

[23] Ricardo Caceffo, Steve Wolfman, Kellogg S Booth, and Rodolfo Azevedo. Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 364–369. ACM, 2016.

[24] Yingjun Cao, Leo Porter, Soohyun Nam Liao, and Rick Ord. Paper or online? a comparison of exam grading techniques. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 99–104, 2019.

[25] David Carson. The abduction of sherlock holmes. *International Journal of Police Science & Management*, 11(2):193–202, 2009.

[26] Salih Çepni, Erol Taş, and Sacit Köse. The effects of computer-assisted material on students' cognitive levels, misconceptions and attitudes towards science. *Computers & Education*, 46(2):192–205, 2006.

[27] Chiu-Liang Chen, Shun-Yin Cheng, and Janet Mei-Chuen Lin. A study of misconceptions and missing conceptions of novice java programmers. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[28] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. Identifying challenging cs1 concepts in a large problem dataset. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 695–700. ACM, 2014.

[29] Sapna Cheryan, Victoria C Plaut, Paul G Davies, and Claude M Steele. Ambient belonging: how stereotypical cues impact gender participation in computer science. *Journal of personality and social psychology*, 97(6):1045, 2009.

[30] Michelene TH Chi. Self-explaining expository texts: The dual processes of generating inferences and repairing mental models. *Advances in instructional psychology*, 5:161–238, 2000.

[31] Michelene TH Chi. Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in cognitive science*, 1(1):73–105, 2009.

[32] Elizabeth F Churchill. Patchwork living, rubber duck debugging, and the chaos monkey. *interactions*, 22(3):22–23, 2015.

[33] Michael Clancy. Misconceptions and attitudes that interfere with learning to program. *Computer science education research*, pages 85–100, 2004.

[34] Herbert H Clark, Susan E Brennan, et al. Grounding in communication. *Perspectives on socially shared cognition*, 13(1991):127–149, 1991.

[35] Paul Cobb and Leslie P Steffe. The constructivist researcher as teacher and model builder. In *A journey in mathematics education research*, pages 19–30. Springer, 2010.

[36] A Cockburn and L Williams. The costs and benefits of pair programming extreme programming examined addison-wesley longman publishing co. *Inc., Boston, MA*, 2001.

[37] Timothy R Colburn and Gary M Shute. Metaphor in computer science. *Journal of Applied Logic*, 6(4):526–533, 2008.

[38] Jere Confrey. Chapter 1: A review of the research on student conceptions in mathematics, science, and programming. *Review of research in education*, 16(1):3–56, 1990.

[39] Michelle Cook, Megan Fowler, Jason O. Hallstrom, Joseph E. Hollingsworth, Tim Schwab, Yu-San Sun, and Murali Sitaraman. Where exactly are the difficulties in reasoning logically about code? experimentation with an online system. In *In Proceedings of ACM 23rd Annual Conference on Innovation and Technology in Computer Science Education*. ACM, 2018.

[40] Michelle Patrick Cook. Visual representations in science education: The influence of prior knowledge and cognitive load theory on instructional design principles. *Science education*, 90(6):1073–1091, 2006.

[41] Graham Cooper and John Sweller. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of educational psychology*, 79(4):347, 1987.

[42] Linda L Cooper and Felice S Shore. Students' misconceptions in interpreting center and variability of data represented via histograms and stem-and-leaf plots. *Journal of Statistics Education*, 16(2), 2008.

[43] Michelle Craig and Andrew Petersen. Student difficulties with pointer concepts in c. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 8. ACM, 2016.

[44] Catherine H Crouch and Eric Mazur. Peer instruction: Ten years of experience and results. *American journal of physics*, 69(9):970–977, 2001.

[45] TG Cummings. Co-constructivism in educational theory and practice, 2001.

[46] Cinda-Sue G Davis and Cynthia J Finelli. Diversity and retention in engineering. *New Directions for Teaching and Learning*, 2007(111):63–71, 2007.

[47] Robert Benjamin Davis. *Learning mathematics: The cognitive science approach to mathematics education*. Greenwood Publishing Group, 1984.

[48] Adrienne Decker. *How students measure up: An assessment instrument for introductory computer science*. PhD thesis, State University of New York at Buffalo, 2007.

[49] Françoise Détienne. Expert programming knowledge: a schema-based approach. In *Psychology of programming*, pages 205–222. Elsevier, 1990.

[50] Françoise Détienne. Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with computers*, 9(1):47–72, 1997.

[51] Edsger W Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[52] Janice A Dole and Gale M Sinatra. Reconceptalizing change in the cognitive construction of knowledge. *Educational psychologist*, 33(2-3):109–128, 1998.

[53] Peter E Doolittle. Complex constructivism: A theoretical model of complexity and cognition. *International Journal of teaching and learning in higher education*, 26(3):485–498, 2014.

[54] Svetlana V Drachova-Strang. *Teaching and assessment of mathematical principles for software correctness using a reasoning concept inventory.* Clemson University, 2013.

[55] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Putting threshold concepts into context in computer science education. In *ACM SIGCSE Bulletin*, volume 38, pages 103–107. ACM, 2006.

[56] Anna Eckerdal and Michael Thuné. Novice java programmers' conceptions of object and class, and variation theory. In *ACM SIGCSE Bulletin*, volume 37, pages 89–93. ACM, 2005.

[57] Stephen H Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, volume 3. Citeseer, 2003.

[58] Stephen H Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin*, 36(1):26–30, 2004.

[59] Michael L Epstein, Amber D Lazarus, Tammy B Calvano, Kelly A Matthews, Rachel A Hendel, Beth B Epstein, and Gary M Brosvic. Immediate feedback assessment technique promotes learning and corrects inaccurate first responses. *The Psychological Record*, 52(2):187–201, 2002.

[60] KA Ericsson. 8: Simon, ha (1984). protocol analysis: Verbal reports as data, 6.

[61] Karl Anders Ericsson and Herbert Alexander Simon. *Protocol analysis: Verbal reports as data, Rev.* the MIT Press, 1993.

[62] William P Eveland Jr and Sharon Dunwoody. Examining information processing on the world wide web using think aloud protocols. *Media Psychology*, 2(3):219–244, 2000.

[63] Richard M Felder and Rebecca Brent. Active learning: An introduction. *ASQ higher education brief*, 2(4):1–5, 2009.

[64] Michael Flanagan and Janice Smith. From playing to understanding: the transformative potential of discourse versus syntax in learning to program. Sense Publishers, 2009.

[65] Ann E Fleury. Parameter passing: the rules the students construct. In *ACM SIGCSE Bulletin*, volume 23, pages 283–286. Citeseer, 1991.

[66] Ann E Fleury. Programming in java: student-constructed rules. In *ACM SIGCSE Bulletin*, volume 32, pages 197–201. ACM, 2000.

[67] Pamela Flores, Nelson Medinilla, and Sonia Pamplona. What do software design students understand about information hiding?: A qualitative case study. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 61–70. ACM, 2014.

[68] Mary Forehand. Bloom's taxonomy. *Emerging perspectives on learning, teaching, and technology*, 41(4):47–56, 2010.

[69] Ellice Ann Forman, Virginia Ramirez-DelToro, Lisa Brown, and Cynthia Passmore. Discursive strategies that foster an epistemic community for argument in a biology classroom. *Learning and Instruction*, 48:32–39, 2017.

[70] National Science Foundation. Computer science for all (csforall:rpp) (nsf:18537), 2018.

[71] National Science Foundation. National science foundation stem education data, 2018.

[72] Barry J Fraser, Herbert J Walberg, Wayne W Welch, and John A Hattie. Syntheses of educational productivity research. *International journal of educational research*, 11(2):147–252, 1987.

[73] Scott Freeman, Sarah L Eddy, Miles McDonough, Michelle K Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences*, 111(23):8410–8415, 2014.

[74] Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pages 173–180. Australian Computer Society, Inc., 2005.

[75] Xun Ge and Susan M Land. Scaffolding students' problem-solving processes in an ill-structured task using question prompts and peer interactions. *Educational Technology Research and Development*, 51(1):21–38, 2003.

[76] Saul Geiser. The growing correlation between race and sat scores: new findings from california. 2015.

[77] David J Gilmore. Expert programming knowledge: a strategic approach. In *Psychology of programming*, pages 223–234. Elsevier, 1990.

[78] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C Loui, and Craig Zilles. Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, 40(1):256–260, 2008.

[79] T. C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clockworks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, 7(2):175–196, 1996.

[80] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 160–168. ACM, 2018.

[81] Mark Guzdial. From science to engineering. *Communications of the ACM*, 54(2):37–39, 2011.

[82] David C Haak, Janneke HilleRisLambers, Emile Pitre, and Scott Freeman. Increased structure and active learning reduce the achievement gap in introductory biology. *Science*, 332(6034):1213–1216, 2011.

[83] Irit Hadar. When intuition and logic clash: The case of the object-oriented paradigm. *Science of Computer Programming*, 78(9):1407–1426, 2013.

[84] Filocha Haslam and David F Treagust. Diagnosing secondary students' misconceptions of photosynthesis and respiration in plants using a two-tier multiple choice instrument. *Journal of biological education*, 21(3):203–211, 1987.

[85] Liana Heitin. Number of students taking ap science exams surges, 2015.

[86] Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. Proof by incomplete enumeration and other logical misconceptions. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 59–70, New York, NY, USA, 2008. ACM.

[87] Richard Higgins, Peter Hartley, and Alan Skelton. The conscientious consumer: Reconsidering the role of assessment feedback in student learning. *Studies in higher education*, 27(1):53–64, 2002.

[88] Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. In *ACM SIGCSE Bulletin*, volume 29, pages 131–134. ACM, 1997.

[89] Md Hossain et al. How to motivate us students to pursue stem (science, technology, engineering and mathematics) careers. *Online Submission*, 2012.

[90] Celia Hoyles, Richard Noss, Ross Adamson, and Sarah Lowe. Programming rules: what do children understand? In *PME conference*, volume 3, pages 3–169, 2001.

[91] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, volume 35, pages 153–156. ACM, 2003.

[92] David L Hu, Lew Lefton, and Peter J Ludovice. Humour applied to stem education. *Systems Research and Behavioral Science*, 34(3):216–226, 2017.

[93] Peter Hubwieser, Johannes Magenheim, Andreas Mühling, and Alexander Ruf. Towards a conceptualization of pedagogical content knowledge for computer science. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 1–8. ACM, 2013.

[94] William Huitt and John Hummel. Piaget's theory of cognitive development. *Educational psychology interactive*, 3(2):1–5, 2003.

[95] Christopher D Hundhausen, Jonathan L Brown, Sean Farley, and Daniel Skarpas. A methodology for analyzing the temporal evolution of novice programs based on semantic components. In *Proceedings of the second international workshop on Computing education research*, pages 59–71. ACM, 2006.

[96] Andrew Hunt. *The pragmatic programmer*. Pearson Education India, 1900.

[97] Robert P Hunting. Clinical interview methods in mathematics education research and practice. *The Journal of Mathematical Behavior*, 16(2):145–165, 1997.

[98] Scott G Isaksen and Donald J Treffinger. Celebrating 50 years of reflective practice: Versions of creative problem solving. *The Journal of Creative Behavior*, 38(2):75–101, 2004.

[99] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013. 999133.

[100] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111. ACM, 2010.

[101] Daniel Kahneman. Maps of bounded rationality: Psychology for behavioral economics. *American economic review*, 93(5):1449–1475, 2003.

[102] Slava Kalyuga, Paul Chandler, Juhani Tuovinen, and John Sweller. When problem solving is superior to studying worked examples. *Journal of educational psychology*, 93(3):579, 2001.

[103] Jeffrey D Karpicke and Janell R Blunt. Retrieval practice produces more learning than elaborative studying with concept mapping. *Science*, 331(6018):772–775, 2011.

[104] Cazembe Kennedy and Eileen T Kraemer. What are they thinking?: Eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, page 7. ACM, 2018.

[105] Cazembe Kennedy and Eileen T Kraemer. Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 224–230, 2019.

[106] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

[107] Päivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In *Proceedings of the second international workshop on Computing education research*, pages 97–108, 2006.

[108] Ole Fogh Kirkeby. Abduktion. *Andersen. H, ed., Videnskabsteori og metodelaere, Fredriksberg: Samfundslitteratur*, pages 122–152, 1994.

[109] Paul A Kirschner. Cognitive load theory: Implications of cognitive load theory on the design of learning, 2002.

[110] Paul A Kirschner, John Sweller, and Richard E Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[111] Leopold E Klopfer, Audrey B Champagne, and Richard F Gunstone. Naive knowledge and science learning. *Research in Science & Technological Education*, 1(2):173–183, 1983.

[112] Sacit Köse. Diagnosing student misconceptions: Using drawings as a research method. *World Applied Sciences Journal*, 3(2):283–293, 2008.

[113] Gyöngyi Kovács and Karen M Spens. Abductive reasoning in logistics research. *International Journal of Physical Distribution & Logistics Management*, 35(2):132–144, 2005.

[114] David R Krathwohl and Lorin W Anderson. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Longman, 2009.

[115] Kenneth L Krause, Robert E Sampsell, and Samuel L Grier. Computer science in the air force academy core curriculum. *ACM SIGCSE Bulletin*, 14(1):144–146, 1982.

[116] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm Sigcse Bulletin*, 37(3):14–18, 2005.

[117] James P Lantolf. *Sociocultural theory and second language learning*, volume 78. Oxford University Press, 2000.

[118] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[119] Aubrey Lawson, Eileen T Kraemer, S Megan Che, and Cazembe Kennedy. A multi-level study of undergraduate computer science reasoning about concurrency. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 210–216, 2019.

[120] Nguyen-Thinh Le. A classification of adaptive feedback in educational systems for programming. *Systems*, 4(2):22, 2016.

[121] Jacqueline Leonard and Danny B Martin. *The brilliance of Black children in mathematics*. IAP, 2013.

[122] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[123] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, volume 36, pages 119–150. ACM, 2004.

[124] Raymond Lister, Anders Berglund, Tony Clear, Joe Bergin, Kathy Garvin-Doxas, Brian Hanks, Lew Hitchner, Andrew Luxton-Reilly, Kate Sanders, Carsten Schulte, et al. Research perspectives on the objects-early debate. *ACM SIGCSE Bulletin*, 38(4):146–165, 2006.

[125] Jennifer A Livingston. Metacognition: An overview. 2003.

[126] Mercedes Lorenzo, Catherine H Crouch, and Eric Mazur. Reducing the gender gap in the physics classroom. *American Journal of Physics*, 74(2):118–122, 2006.

[127] Andrew Luxton-Reilly, Brett A Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Jacqueline Whalley, et al. Developing assessments to determine mastery of programming fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, pages 47–69. ACM, 2018.

[128] Philip Machanick. The abstraction-first approach to data abstraction and algorithms. *Computers & Education*, 31(2):135–150, 1998.

[129] Sandra Madison and James Gifford. Modular programming: novice misconceptions. *Journal of Research on Technology in Education*, 34(3):217–229, 2002.

[130] Lauren E Margulieux and Richard Catrambone. Improving problem solving with subgoal labels in expository text and worked examples. *Learning and Instruction*, 42:58–71, 2016.

[131] Richard E Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR)*, 13(1):121–141, 1981.

[132] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. ACM, 2001.

[133] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

[134] Jan H. F. Meyer and Ray Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):373–388, Apr 2005.

[135] Joan Middendorf and Alan Kalish. The "change-up" in lectures. In *Natl. Teach. Learn. Forum*, volume 5, pages 1–5, 1996.

[136] Craig S Miller and Amber Settle. Some trouble with transparency: An analysis of student errors with object-oriented python. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 133–141. ACM, 2016.

[137] Seyed Yaghoub Mousavi, Renae Low, and John Sweller. Reducing cognitive load by mixing auditory and visual presentation modes. *Journal of educational psychology*, 87(2):319, 1995.

[138] Engineering National Academies of Sciences, Medicine, et al. *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press, 2018.

[139] Greg L Nelson, Benjamin Xie, and Andrew J Ko. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 2–11. ACM, 2017.

[140] David J Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education*, 31(2):199–218, 2006.

[141] Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM, 2002.

[142] Steve Olson and Donna Gerardi Riordan. Engage to excel: Producing one million additional college graduates with degrees in science, technology, engineering, and mathematics. report to the president. *Executive Office of the President*, 2012.

[143] Anthony J Onwuegbuzie, R Burke Johnson, and Kathleen MT Collins. Assessing legitimation in mixed research: a new framework. *Quality & Quantity*, 45(6):1253–1271, 2011.

[144] Rachel Or-Bach and Ilana Lavy. Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin*, 36(2):82–86, 2004.

[145] Paul Orsmond, Stephen Merry, and Kevin Reiling. The use of exemplars and formative feedback when using student derived marking criteria in peer and self-assessment. *Assessment & Evaluation in Higher Education*, 27(4):309–323, 2002.

[146] Alex Osborn. *Applied Imagination-Principles and Procedures of Creative Writing*. Read Books Ltd, 2012.

[147] Roger J Osborne and Merlin C Wittrock. Learning science: A generative process. *Science education*, 67(4):489–508, 1983.

[148] Miranda C Parker, Mark Guzdial, and Shelly Engleman. Replication, validation, and use of a language independent cs1 knowledge assessment. In *Proceedings of the 2016 ACM conference on international computing education research*, pages 93–101. ACM, 2016.

[149] David Lorge Parnas. Information distribution aspects of design methodology. 1971.

[150] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[151] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. *Communications of the ACM*, 63(1):91–98, 2019.

[152] Wolfgang Paul and Jan Vahrenhold. Hunting high and low: instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 29–34. ACM, 2013.

[153] Roy D Pea. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1):25–36, 1986.

[154] Joan Peckham, Lisa L Harlow, David A Stuart, Barbara Silver, Helen Mederer, and Peter D Stephenson. Broadening participation in computing: issues and challenges. In *ACM SIGCSE Bulletin*, volume 39, pages 9–13. ACM, 2007.

[155] Nancy Pennington, Adrienne Y Lee, and Bob Rehder. Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, 10(2):171–226, 1995.

[156] David Perkins. The many faces of constructivism. *Educational leadership*, 57(3):6–11, 1999.

[157] David N Perkins, Steve Schwartz, and Rebecca Simmons. Instructional strategies for the problems of novice programmers. 1988.

[158] DN Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In *first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, 1986.

[159] William G. Perry Jr. *Forms of Intellectual and Ethical Development in the College Years: A Scheme*. Jossey-Bass Publishers, San Francisco, CA, 1999.

[160] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*, 2015.

[161] Peter Pirolli and Stuart Card. Information foraging. *Psychological review*, 106(4):643, 1999.

[162] Leo Porter, Cynthia Bailey Lee, Beth Simon, and Daniel Zingaro. Peer instruction: do students really learn from peer discussion in computing? In *Proceedings of the seventh international workshop on Computing education research*, pages 45–52. ACM, 2011.

[163] George J Posner, Kenneth A Strike, Peter W Hewson, and William A Gertzog. Accommodation of a scientific conception: Toward a theory of conceptual change. *Science education*, 66(2):211–227, 1982.

[164] Sarah Monisha Pulimood and Ursula Wolz. Problem solving in community: a necessary shift in cs pedagogy. *ACM SIGCSE Bulletin*, 40(1):210–214, 2008.

[165] Mary A Pyc, Pooja K Agarwal, and Henry L Roediger III. Test-enhanced learning. 2014.

[166] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

[167] Noa Ragonis and Mordechai Ben-Ari. A long-term investigation of the comprehension of oop concepts by novices. 2005.

[168] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[169] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[170] Henry L Roediger III and Andrew C Butler. The critical role of retrieval practice in long-term retention. *Trends in cognitive sciences*, 15(1):20–27, 2011.

[171] Kenneth J Rothman. No adjustments are needed for multiple comparisons. *Epidemiology*, pages 43–46, 1990.

[172] Philip M Sadler, Gerhard Sonnert, Harold P Coyle, Nancy Cook-Smith, and Jaimie L Miller. The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal*, 50(5):1020–1049, 2013.

[173] Kate Sanders and Robert McCartney. Threshold concepts in computing: past, present, and future. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, pages 91–100. ACM, 2016.

[174] Kate Sanders and Lynda Thomas. Checklists for grading object-oriented cs1 programs: concepts and misconceptions. *ACM SIGCSE Bulletin*, 39(3):166–170, 2007.

[175] Erik P Schmidt. Postsecondary enrollment before, during, and since the great recession. population characteristics. current population reports. p20-580. *US Census Bureau*, 2018.

[176] Andreas Schwill. Fundamental ideas of computer science. *Bull. European Assoc. for Theoretical Computer Science*, 53, 1994.

[177] Lee Shulman. Knowledge and teaching: Foundations of the new reform. *Harvard educational review*, 57(1):1–23, 1987.

[178] Lee S Shulman. Those who understand: Knowledge growth in teaching. *Educational researcher*, 15(2):4–14, 1986.

[179] Herbert A Simon. Information-processing theory of human problem solving. *Handbook of learning and cognitive processes*, 5:271–295, 1978.

[180] Rebecca A Simon, Mark W Aulls, Helena Dedic, Kyle Hubbard, and Nathan C Hall. Exploring student persistence in stem programs: a motivational model. *Canadian Journal of Education*, 38(1):n1, 2015.

[181] Teemu Sirkiä and Juha Sorva. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 19–28. ACM, 2012.

[182] Burrhus Frederic Skinner. *About behaviorism*. Vintage, 2011.

[183] D Sleeman, Ralph T Putnam, Juliet Baxter, and Laiani Kuspa. An introductory pascal class: A case study of students' errors. *Teaching and Learning Computer Programming: Multiple Research Perspectives. RE Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates*, pages 237–257, 1988.

[184] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Workshop on Software and Performance*, volume 17, pages 127–136. Ottawa, Canada, 2000.

[185] John P Smith III, Andrea A Disessa, and Jeremy Roschelle. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The journal of the learning sciences*, 3(2):115–163, 1994.

[186] Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[187] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.

[188] James C Spohrer, Elliot Soloway, and Edgar Pope. A goal/plan analysis of buggy pascal programs. *Human–Computer Interaction*, 1(2):163–207, 1985.

[189] Claude M Steele. Race and the schooling of black americans. *The Atlantic Monthly*, 269(4):68–78, 1992.

[190] Claude M Steele. A threat in the air: How stereotypes shape intellectual identity and performance. *American psychologist*, 52(6):613, 1997.

[191] Anselm Strauss and Juliet Corbin. Grounded theory methodology. *Handbook of qualitative research*, 17:273–85, 1994.

[192] Anselm Strauss and Juliet M Corbin. *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications, Inc, 1990.

[193] John Stuart and RJD Rutherford. Medical student concentration during lectures. *The lancet*, 312(8088):514–516, 1978.

[194] John Sweller. Cognitive load theory, learning difficulty, and instructional design. *Learning and instruction*, 4(4):295–312, 1994.

[195] John Sweller. Cognitive load theory. In *Psychology of learning and motivation*, volume 55, pages 37–76. Elsevier, 2011.

[196] John Sweller and Graham A Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction*, 2(1):59–89, 1985.

[197] Iris Tabak and Eric Baumgartner. The teacher as partner: Exploring participant structures, symmetry, and identity work in scaffolding. *Cognition and Instruction*, 22(4):393–429, 2004.

[198] C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. Computer science concept inventories: past and future. *Computer Science Education*, 24(4):253–276, 2014.

[199] Steven S Taylor, Dalmar Fisher, and Ronald L Dufresne. The aesthetics of management storytelling: a key to organizational learning. *Management Learning*, 33(3):313–330, 2002.

[200] Steven R Terrell. Mixed-methods research methodologies. *Qualitative report*, 17(1):254–280, 2012.

[201] Allison Elliott Tew and Mark Guzdial. The fcs1: a language independent assessment of cs1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 111–116. ACM, 2011.

[202] CORPORATE The Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal on Educational Resources in Computing (JERIC)*, 1(3es):1–es, 2001.

[203] Lynda A Thomas, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, and Carol Zander. In the liminal space: software design as a threshold skill. *Practice and Evidence of the Scholarship of Teaching and Learning in Higher Education*, 12(2):333–351, 2017.

[204] Benjy Thomasson, Mark Ratcliffe, and Lynda Thomas. Identifying novice difficulties in object oriented design. *ACM SIGCSE Bulletin*, 38(3):28–32, 2006.

[205] Julia D Thompson, Geoffrey L Herman, Travis Scheponik, Linda Oliva, Alan Sherman, Ennis Golaszewski, Dhananjay Phatak, and Kostantinos Patsourakos. Student misconceptions about cybersecurity concepts: Analysis of think-aloud interviews. *Journal of Cybersecurity Education, Research and Practice*, 2018(1):5, 2018.

[206] Welko Tomic. Behaviorism and cognitivism in education. *Psychology*, 30(3/4):38–46, 1993.

[207] Sheng-Chau Tseng and Chin-Chung Tsai. On-line peer assessment and the role of the peer feedback: A study of high school computer course. *Computers & Education*, 49(4):1161–1174, 2007.

[208] Evgenia Vagianou. Program working storage: a beginner's model. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 69–76. ACM, 2006.

[209] Lev Vygotsky. Interaction between learning and development. *Readings on the development of children*, 23(3):34–41, 1978.

[210] Barry J Wadsworth. *Piaget's theory of cognitive and affective development: Foundations of constructivism*. Longman Publishing, 1996.

[211] Noreen M Webb, Philip Ender, and Scott Lewis. Problem-solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal*, 23(2):243–261, 1986.

[212] Eric W Weisstein. Bonferroni correction. 2004.

[213] Chris Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 90–95. ACM, 2015.

[214] Laurie Williams and Robert Kessler. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[215] Laurie Williams, Eric Wiebe, Kai Yang, Miriam Ferzli, and Carol Miller. In support of pair programming in the introductory computer science course. *Computer Science Education*, 12(3):197–212, 2002.

[216] Leon E Winslow. Programming pedagogy—a psychological overview. *ACM Sigcse Bulletin*, 28(3):17–22, 1996.