UNDERSTANDING AND ADDRESSING MISCONCEPTIONS IN INTRODUCTORY PROGRAMMING: A DATA-DRIVEN APPROACH

by

Yizhou Qian

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Curriculum & Instruction

West Lafayette, Indiana

May 2018

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

To Grace and Ada

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. James Lehman, who has offered me great support and guidance during my journey to the PhD degree. You are like an encyclopedia that knows all the answers to my questions.

I also would like to thank my committee members, Dr. Tim Newby, Dr. Susanne Hambrusch, and Dr. Aman Yadav, who have provided critical and valuable feedback to my research and given me beneficial advice to my career.

Next, I would like to thank my wife, Panpan Zou, for all her sacrifice and support during my PhD study.

Last but not least, I would like to thank all the friends I have met here at Purdue. I have learned a lot from you, both academic and non-academic.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Author: Qian, Yizhou. PhD
Institution: Purdue University
Degree Received: May 2018
Title: Understanding and Addressing Misconceptions in Introductory Programming: A Data-Driven Approach
Major Professor: James Lehman

With the expansion of computer science (CS) education, CS teachers in K-12 schools should be cognizant of student misconceptions and be prepared to help students establish accurate understanding of computer science and programming. This exploratory design-based research (DBR) study implemented a data-driven approach to identify secondary school students' misconceptions using both their compilation and test errors and provide targeted feedback to promote students' conceptual change in introductory programming. Research subjects were two groups of high school students enrolled in two sections of a Java-based programming course in a 2017 summer residential program for gifted and talented students. This study consisted of two stages. In the first stage, students of group 1 took the introductory programming class and used an automated learning system, Mulberry, which collected data on student problem-solving attempts. Data analysis was conducted to identify common programming errors students demonstrated in their programs and relevant misconceptions. In the second stage, targeted feedback to address these misconceptions was designed using principles from conceptual change and feedback theories and added to Mulberry. When students of group 2 took the same introductory programming class and solved programming problems in Mulberry, they received the targeted feedback to address their misconceptions. Data analysis was conducted to assess how the feedback affected the evolution of students' (mis)conceptions.

Using students' erroneous solutions, 55 distinct compilation errors were identified, and 15 of them were categorized as common ones. The 15 common compilation errors accounted for 92% of all compilation errors. Based on the 15 common compilation errors, three underlying student misconceptions were identified, including deficient knowledge of fundamental Java program structure, misunderstandings of Java expressions, and confusion about Java variables. In addition, 10 common test errors were identified based on nine difficult problems. The results showed that 54% of all test errors were related to the difficult problems, and the 10 common test errors accounted for 39% of all test errors of the difficult problems. Four common student misconceptions were identified based on the 10 common test errors, including misunderstandings of Java input, misunderstandings of Java output, confusion about Java operators, and forgetting to consider special cases.

Both quantitative and qualitative data analysis were conducted to see whether and how the targeted feedback affected students' solutions. Quantitative analysis indicated that targeted feedback messages enhanced students' rates of improving erroneous solutions. Group 2 students showed significantly higher improvement rates in all erroneous solutions and solutions with common errors compared to group 1 students. Within group 2, solutions with targeted feedback messages resulted in significantly higher improvement rates compared to solutions without targeted feedback messages. Results suggest that with targeted feedback messages students were more likely to correct errors in their code. Qualitative analysis of students' solutions of four selected cases determined that students of group 2, when improving their code, made fewer intermediate incorrect solutions than students in group 1. The targeted feedback messages appear to have helped to promote conceptual change.

The results of this study suggest that a data-driven approach to understanding and addressing student misconceptions, which is using student data in automated assessment systems, has the potential to improve students' learning of programming and may help teachers build better understanding of their students' common misconceptions and develop their pedagogical content knowledge (PCK). The use of automated assessment systems with misconception identification components may be helpful in pre-college introductory programming courses and so is encouraged as K-12 CS education expands. Researchers and developers of automated assessment systems should develop components that support identifying common student misconceptions using both compilation and non-compilation errors. Future research should continue to investigate the use of targeted feedback in automated assessment systems to address students' misconceptions and promote conceptual change in computer science education.

CHAPTER 1: INTRODUCTION

The development of computing technology and its role in driving innovation and economic development in the 21st century has brought the need for expanding computer science (CS) education (Webb et al., 2017). Many countries have included computer science courses in their K-12 curriculum. In the U.S., Advanced Placement (AP) CS Principles, a new introductory computer science course for high school students has been developed (desJardins, 2015). In the U.K., computer science has become mandatory for students in K-12 schools (Brown, Sentance, Crick, & Humphreys, 2014). In New Zealand, computer science has been a mainstream subject since 2011 (Bell, Andreae, & Robins, 2014). Many other countries, such as Canada, Israel, Poland, and so forth, have created new or improved existing computer science curricula (Webb et al., 2017). With the expansion of computer science education, CS teachers in K-12 schools should be cognizant of student misconceptions and be prepared to help students establish accurate understanding of computer science and programming. This exploratory design-based research study implemented a data-driven approach to identify secondary school students' misconceptions and provide targeted feedback to promote students' conceptual change in introductory programming.

Student Misconceptions in Introductory Programming

Introductory CS courses are difficult for beginners (Guzdial, 2015; McCracken et al., 2001), and students often exhibit misconceptions that impede their learning of introductory programming (Altadmri & Brown, 2015; Sorva, 2013). Qian and Lehman (2017) summarized literature regarding common misconceptions and other difficulties in introductory programming. For instance, variables are a very basic concept in most of the programming languages, but novices may mistakenly believe that the computer understands variables by the English meanings of their names, even though variable names are arbitrary (Kaczmarczyk, Petrick, East, & Herman, 2010; Sleeman, Putnam, Baxter, & Kuspa, 1986). Sequential execution of code is another challenging concept for beginners (du Boulay, 1986; Simon, 2011). For instance, students may mistakenly believe that when the Boolean expression of a conditional statement becomes true, even if this occurs twenty lines below the conditional statement, the program will go back and execute the code in that conditional block (Pea, 1986). High-level concepts such as classes, objects, instances, and their relationships in object-oriented programming (OOP) also often confuse students (Guzdial, 1995; Holland, Griffiths, & Woodman, 1997; Ragonis & Ben-Ari, 2005; Sorva, 2013).

Students in introductory programming courses may exhibit syntax errors, logic errors, and other difficulties when writing programs to solve problems. For instance, novice students often make syntactic mistakes in their code, such as mismatching parentheses, missing semicolons, failing to declare a variable, using malformed Boolean expressions, mistakenly using the assignment operator (=) instead of the comparison operator (==), and so forth (Altadmri & Brown, 2015; Jackson, Cobb, & Carver, 2005; Sirkia & Sorva, 2012). In addition, they usually lack well-established programming strategies (Clancy & Linn, 1999; Davies, 1993; Lister, Simon, Thompson, Whalley, & Prasad, 2006; Sajaniemi & Prieto, 2005; Soloway, 1986) and then face difficulties with planning, composing, and debugging programs, including failing to understand and decompose the task (Muller, 2005; Robins, Haden, & Garner, 2006), forgetting to test boundary conditions and unexpected cases (Fisler, Krishnamurthi, & Siegmund, 2016; Sajaniemi & Kuittinen, 2005; Spohrer & Soloway, 1986), and inappropriately tracing their code and locating errors (Ben-David Kolikant & Mussai, 2008; Fitzgerald et al., 2008; McCauley et al., 2008).

Qian and Lehman (2017) described factors that may contribute to students' misconceptions in learning to program. Major factors that contribute to students' misconceptions include interference caused by prior knowledge (Clancy, 2004; Miller, 2014) and flawed mental models of computer operation (Guzdial, 2015; Sorva, 2013). Novice students may mistakenly use concepts they learned in math to understand programming concepts (e.g., variables), which look similar but mean something quite different (Clancy, 2004; Qian & Lehman, 2017). As most programming languages are natural-language-based, students' existing knowledge of natural language may hinder their construction of the meanings of programming concepts (Bruckman & Edwards, 1999; du Boulay, 1986; Miller, 2014). In addition, unlike experts, beginners' conceptual knowledge is often fragmentary and not organized into meaningful patterns (Clancy & Linn, 1999; Lister, 2011; McCauley et al., 2008; Sajaniemi & Prieto, 2005; Whalley et al., 2006). Thus, they may only be able to understand programs in a line-by-line manner and then fail to holistically evaluate and properly debug a program (Ben-David Kolikant & Mussai, 2008;

Lister et al., 2006). Students in introductory programming courses also often hold flawed mental models of the notional machine, which refers to an abstract computer that executes code in the programmer's mind (du Boulay, 1986; Guzdial, 2015; Sorva, 2013). Without correct understanding of the notional machine, a student may fail to understand the sequential execution of statements (du Boulay, 1986; Simon, 2011).

Student misconceptions can interfere with learning, and a variety of factors may contribute to these inaccurate understandings (Clancy, 2004; Qian & Lehman, 2017; Smith, diSessa, & Roschelle, 1994). While previous studies have cataloged a broad range of student misconceptions including syntax errors and other difficulties caused by misconceptions, most of them have focused on post-secondary students (e.g., Altadmri & Brown, 2015; Jackson et al., 2005; Sirkia & Sorva, 2012). As CS education has been expanding into K-12 schools, more information is needed to understand misconceptions among pre-college learners who take introductory programming courses. This study investigated misconceptions among secondary school students taking an introductory programming course.

Theoretical Framework

In the learning of science, conceptions refer to students' understandings of academic concepts (Taber, 2013). Misconceptions are problematic conceptions held by students which are inconsistent with normative conceptions and often entrenched (Clement, 1993; Smith et al., 1994; Taber, 2013). Similarly, in the learning of programming, student misconceptions are students' deficient or erroneous understandings of programming concepts (Qian & Lehman, 2017; Sorva, 2013; Taber, 2013). In previous literature, a variety of terms have been used to describe students' inaccurate understandings in learning to program, such as "misconceptions" (Sorva, 2013), "difficulties" (du Boulay, 1986), "errors" (Sleeman et al., 1986), "bugs" (Pea, 1986), "mistakes" (Altadmri & Brown, 2015), and so forth. With these different terms, researchers have discussed students' syntax errors in the code, misunderstandings of programming concepts, difficulties in writing and debugging programs, and so on (Sorva, 2013). While these different misunderstandings are often lumped together, qualitative differences exist between a simple syntax error in a loop statement, conceptual misunderstandings of loops, and challenges of using loop constructs to solve problems. However, these difficulties are related to each other, and problems of students' conceptual understandings are the pivot that may lead to syntactic errors, logic errors, and other difficulties (Bayman & Mayer, 1988; de Raadt, 2008; Ebrahimi, 1994; Lopez, Whalley, Robbins, & Lister, 2008; Qian & Lehman, 2017). Thus, to help students succeed in introductory programming, it is vital to understand and address student misconceptions.

Conceptual Change Theories

In science and mathematics education, researchers and educators have developed conceptual change theories to address student misconceptions. Conceptual change denotes the process through which learners' existing (mis)conceptions develop into intended normative conceptions (Duit & Treagust, 2003; Vosniadou & Skopeliti, 2014). Conceptual change theories inform the process of modifying student misconceptions to help students establish normative understandings of the academic concepts to be learned (Vosniadou & Skopeliti, 2014). Researchers of conceptual change theories share the ideas that (a) learners' own pre-instructional conceptions (also called naïve knowledge) are based on their daily experience; (b) learners' existing knowledge has an impact on the acquisition of new knowledge; and (c) student misconceptions are often entrenched and conceptual change is time consuming (Özdemir & Clark, 2007; Taber, 2013).

Two conflicting theoretical perspectives, revolutionary conceptual change and evolutionary conceptual change, have emerged over the decades of research (Abimbola, 1988; Özdemir & Clark, 2007; Taber, 2013). The revolutionary conceptual change perspective posits that learners' existing naïve knowledge is organized in a theory-like manner, and learners use their naïve theories to interpret and construct new concepts (Özdemir & Clark, 2007; Posner, Strike, Hewson, & Gertzog, 1982). Thus, learners' existing misconceptions are a potential barrier to new learning, and conceptual change is a revolutionary process that replaces learners' naïve theory-like knowledge structures with intended scientific conceptions. According to the revolutionary conceptual change perspective, successful instruction needs to help students confront their misconceptions by presenting the academic concept to students in a way that produces cognitive conflicts, and then help students abandon their misconceptions and adopt the new conceptions (Abimbola, 1988; Posner et al., 1982).

In contrast, the evolutionary conceptual change perspective postulates that learners' prior naïve knowledge consists of relatively unstructured collections of quasi-independent elements

(Abimbola, 1988; diSessa, 1993). From this viewpoint, conceptual change is an evolutionary process of correcting and enhancing existing knowledge elements and establishing and refining the relationships among conceptions. Therefore, learners' existing (mis)conceptions should be considered as resources for constructing new concepts, and the purpose of instruction is to reconcile students' prior (mis)conceptions with new learning, rather than replacing them (Abimbola, 1988; diSessa, 2013, 2014).

While debate between the two perspectives is ongoing (see diSessa, 2013 and Vosniadou, 2013), the current trend in conceptual change research has shown convergence (Vosniadou & Skopeliti, 2014). Researchers agree that evolutionary conceptual change is a prerequisite of revolutionary conceptual change (Taber, 2013; Vosniadou & Skopeliti, 2014), and success in conceptual change requires tracking the development of learners' (mis)conceptions using real-time data of learning (diSessa, 2014; Vosniadou, 2013). With precise understanding of the nature and current status of student (mis)conceptions, instructors can choose proper strategies for accomplishing conceptual change, such as "directly challenging student conceptions," "ignoring them and simply teaching the canonical ideas," or "seeing learners' conceptions as useful (or necessary) starting points that need to be modified over time through a multistage conceptual trajectory" (Taber, 2014, p. 40). While conceptual change theories have been widely adopted to understand the development of student knowledge in math and science (Vosniadou & Skopeliti, 2014), they have received relatively little attention in CS education to date (Qian & Lehman, 2017; Sorva, 2012). This study applied conceptual change theories to students' learning of introductory programming in computer science.

Models of Feedback

Feedback is essential to help learners successfully construct new knowledge. Historically, researchers have defined feedback from three different perspectives: feedback as motivator, feedback as reinforcement, and feedback as information (Kulhavy & Wager, 1993). According to the motivational viewpoint, feedback is a motivator or incentive for enhancing learning performance. However, the mix of motivation and feedback makes it difficult to conceptualize how feedback works (Kulhavy & Wager, 1993). The feedback-as-reinforcement perspective posits that feedback producing a satisfying effect is likely to make the response repeated in the future. This idea was derived from E. L. Thorndike's Law of Effect and greatly developed by B.

F. Skinners' study of programmed instruction (Kulhavy & Wager, 1993; Mory, 2004). These researchers believed that telling the learner his or her answer is correct would increase the probability of making the same right response in the future. Hence, studies of the feedback-as-reinforcement perspective mainly focused on learners' correct responses and often ignored errors (Kulhavy, 1977). Instead of concentrating on correct responses, the feedback-as-information perspective emphasizes learners' erroneous responses and considers feedback as information for correcting learners' errors and misunderstandings (Butler & Winne, 1995; Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Shute, 2008). While researchers nowadays do not deny that feedback in education may lead to changes in learners' motivation and reinforcement of learning, they agree that feedback in essence is information for facilitating learning (Hattie & Gan, 2011; Hattie & Timperley, 2007; Shute, 2008). More specifically, feedback is information provided by an agent to change learners' thinking or behavior for the purpose of enhancing learning (Hattie & Timperley, 2007; Shute, 2008).

A number of factors may influence the effect of feedback on learning, including timing, complexity, and sources of feedback (Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Van der Kleij, Feskens, & Eggen, 2015), and researchers have developed different models of feedback to explain how feedback facilitates learning and provide guidelines for designing effective feedback. Well-known models of feedback include the certitude model with a focus on learner response confidence (Kulhavy & Stock, 1989), the five-stage model emphasizing learners' mindful reflection (Bangert-Drowns, Kulik, Kulik, & Morgan, 1991), and the connectionist model concentrating on the retention of initial lesson responses (Clariana, Wagner, & Murphy, 2000). Each of these models mainly addresses one factor that may contribute to feedback's effects, such as confidence (Kulhavy & Stock, 1989), self-regulation (Bangert-Drowns et al., 1991; Butler & Winne, 1995), and timing (Clariana et al., 2000).

Recently, a more inclusive model, the visibility model of feedback, with an emphasis on visualizing learners' current knowledge states, has been developed and widely accepted (Hattie & Gan, 2011; Hattie & Timperley, 2007). According to the visibility model, feedback reduces "the discrepancy between what is understood and what is aimed to be understood" (Hattie & Gan, 2011, p. 257-258). The crux of feedback design is to make the discrepancy visible to both the instructor and the learner. The visibility means that effective feedback needs to answer three major questions: "Where am I going?," "How am I going?," and "Where to next?" (Hattie &

Timperley, 2007). In other words, procedures for designing effective feedback include (1) clearly describing the desired learning outcomes, (2) precisely analyzing learners' current knowledge states, and (3) identifying the discrepancy between the current states and the intended outcomes and providing information for reducing the discrepancy and enhancing learning (Hattie & Gan, 2011; Hattie & Timperley, 2007). From this point of view, the problem of traditional feedback design is that it neglects to examine learners' current (mis)conceptions but simply provides corrective information for fixing superficial learner errors (Hattie & Gan, 2011). The new model of feedback requires scrutinizing learners' erroneous responses, to grasp their positions on the trajectory towards the success of the learning goals, and provide corrective information targeted at addressing student misconceptions (Hattie & Gan, 2011; Hattie & Timperley, 2007).

Data-driven Feedback for Conceptual Change

The evolutionary conceptual change theory offers a framework for understanding student misconceptions and the development of ways to address student (mis)conceptions. The visibility model provides a framework for designing feedback for addressing misconceptions and promoting conceptual change. They both emphasize the importance of understanding current states of learner knowledge and tracking the evolution of student (mis)conceptions using learner data. This study adopted the evolutionary conceptual change theory and the visibility model of feedback as the theoretical framework for understanding and addressing student misconceptions. While previous studies in computer science education have discussed student misconceptions from a variety of perspectives, little work has drawn on our understanding of evolutionary conceptual change and appropriate use of data-driven feedback to promote conceptual change (Qian & Lehman, 2017).

Instructional Approaches and Tools for Addressing Student Misconceptions

In computer science education, researchers and educators have developed various instructional approaches to address students' misconceptions in introductory programming (Qian & Lehman, 2017). Using program examples in instruction is one effective approach. Previous research has revealed that using worked-out examples in instruction can improve students' performance in solving problems (Ginat, Shifroni, & Menashe, 2011). In addition, asking students to comprehend and explain example programs can help to disclose students' misconceptions and develop their program composing and debugging skills (McCauley et al., 2008; Teague & Lister, 2014; Vainio & Sajaniemi, 2007). Another approach is explicitly teaching programming strategies to help students reduce cognitive load in programming and improve their ability to decompose and solve problems (de Raadt, 2008; Muller, Ginat, & Haberman, 2007). Some researchers advocate using another approach, a concept inventory (Goldman et al., 2010; Taylor et al., 2014; Tew, 2010). A concept inventory is an assessment aimed at evaluating students' understanding of a group of concepts (Goldman et al., 2010; Tew, 2010). Using a concept inventory to evaluate students' understanding of key programming concepts enables instructors to identify common misconceptions students have and then improve their instruction based on the misconceptions (Taylor et al., 2014). Another instructional approach is Peer Instruction (Porter, Lee, & Simon, 2013), which focuses on engaging students in active learning of new concepts. It includes three steps: answering a question individually, having discussions with peers, and reconsidering the question again (Simon, Kohanfars, Lee, Tamayo, & Cutts, 2010). Previous research on Peer Instruction indicated that it can effectively improve students' learning performance in introductory programming (Porter et al., 2013; Simon et al., 2010).

In addition to instructional approaches, researchers and educators have also developed instructional tools to address students' misconceptions in introductory programming, such as novice programming environments that help to prevent syntax errors (Kelleher & Pausch, 2005; Resnick et al., 2009), debugging tools that improve students' understanding of their errors (Becker et al., 2016; Ko & Myers, 2005), and visualization tools that illustrate programming concepts and program execution (Guo, 2013; Sorva, Karavirta, & Malmi, 2013). Of particular interest for this study is the development of automated assessment systems that can automatically assess students' programs and provide immediate feedback to help students learn (De-La-Fuente-Valentín, Pardo, & Delgado Kloos, 2013; Douce, Livingstone, & Orwell, 2005; Gerdes, Heeren, Jeuring, & van Binsbergen, 2017). One way to provide feedback is to manually integrate enhanced compiler error messages (Becker, 2016). Decaf is a tool that can provide novice-friendly feedback messages by enhancing raw Java compiler error messages (Becker, 2016). In a study of using Decaf to teach a Java-based CS1 class, Becker (2016) reported that the group receiving feedback messages made 32% fewer errors than the group only seeing the raw Java compiler error messages. Another way to provide feedback is designing algorithms to automatically diagnose students' programs and personalize the feedback for students (Barnes & Stamper, 2010; Rivers & Koedinger, 2017; Xu & Chee, 2003). However, such systems typically only work on simple programming problems and provide feedback based on students' errors in code rather than misconceptions (Gerdes et al., 2017).

This study used Mulberry, an automated assessment system designed for Java learners, to support learning. While many automated assessment systems have been developed and tested by researchers, most systems to date either provide feedback based on compiler errors (e.g., Becker, 2016) or provide direct feedback for correcting simple errors in code (e.g., Gerdes et al., 2017). This study focuses on analyzing both compilation and test errors in students' programs to understand student misconceptions and provide feedback targeted at promoting conceptual change.

Purpose of the Study

The purpose of this exploratory design-based research study was to examine secondary school students' common misconceptions in introductory programming using both their compilation and test errors and investigate how feedback affected the evolution of students' (mis)conceptions using a data-driven approach. The following research questions guided the study:

1. What are secondary school students' common misconceptions in introductory programming?
2. How does feedback, developed to promote conceptual change, affect students' (mis)conceptions?

Research Design

A two-stage, exploratory, design-based study was implemented. Research subjects in this study were two groups of high school students enrolled in two sections of a Java-based programming course as part of a 2017 summer residential program for gifted and talented students. Students were identified as high ability according to the rules of the residential program.

In the Java-based programming course, Mulberry, a programming learning system designed for Java learners, was integrated into instruction. Mulberry has a pool of 51 programming problems, and students are required to write short programs to produce the correct output to solve the problems. Mulberry automatically assesses students' solutions by comparing the output of their programs with the expected output. When a student submits a program producing the incorrect output, he or she receives immediate feedback from the system and can try as many times as needed until his or her solution is correct. Mulberry collects all the programs from students when they attempt to solve the problems.

For the first group, when students had errors in their solutions, they were told that errors existed in their code and were encouraged to try again. After the first group's course ended, data analysis was conducted to identify common programming errors students demonstrated in their programs and relevant misconceptions to answer RQ 1. Then, targeted feedback to address these misconceptions was designed using principles from conceptual change and feedback theories (diSessa, 2014; Hattie & Gan, 2011; Vosniadou & Skopeliti, 2014) and added to Mulberry. When students of the second group solved problems in Mulberry and submitted solutions producing incorrect output, they received the targeted feedback to address their misconceptions. After the second group's course ended, data analysis was conducted to assess how feedback affected the evolution of students' (mis)conceptions to answer RQ 2.

Organization of the Remaining Chapters

The remaining chapters of this dissertation are organized as follows. Chapter 2 provides a review of the relevant literature that frames this study. The reviewed literature discusses three major topics: misconceptions and conceptual change theories, feedback in education, and student misconceptions in introductory programming. Chapter 3 provides a description of the research methods, participants, and procedures. Chapter 4 presents the results of the study. Chapter 5 discusses the results, conclusions, and implications.

CHAPTER 2: LITERATURE REVIEW

The goal of this study was to investigate student misconceptions in introductory programming and the effects of targeted feedback for addressing misconceptions and promoting conceptual change. This chapter first reviews relevant literature on misconceptions and feedback in education to build the theoretical framework for this study. Second, studies about student misconceptions in introductory programming are reviewed.

Misconceptions and Conceptual Change Theories

Student misconceptions have gained attention from researchers and educators in science education since 1980s. A variety of conceptual change theories have been developed to explain and address student misconceptions. This section reviews studies about misconceptions and conceptual change theories.

Definitions of Misconceptions

In the research of student conceptions in science, the use of terminology is problematic (Abimbola, 1988; Taber, 2013). Researchers often mistakenly assume that people understand the same term in the same way (Taber, 2013). Hence, it is important to clearly define the terms of the study in the first place. In science education, conceptions refer to students' understandings of academic concepts (Taber, 2013). Students' development of conceptual understanding may vary across individuals, because students may be taught the same academic concept (e.g., energy) in different classes but would each build their own personal conceptions (e.g., of energy). Misconceptions are problematic conceptions held by students which are inconsistent with normative conceptions and often entrenched (Clement, 1993; Smith et al., 1994; Taber, 2013). While a myriad of studies regarding student misconceptions in science have been conducted over the past several decades, nowadays researchers are still using different terms to describe students' understandings of academic concepts that are at odds with scientific ones, such as misconceptions, alternative conceptions, alternative frameworks, naïve knowledge, intuitive knowledge, and others (Clement, 1993; diSessa, 2014; Klopfer, Champagne, & Gunstone, 1983; Özdemir & Clark, 2007; Taber, 2014). Though some minor differences exist among these terms, generally they are considered as synonyms (Taber, 2014). This study uses the term misconception as it is widely used by computing education researchers (Sorva, 2013).

Conceptual Change Theories

In science and mathematics education, researchers and educators have developed conceptual change theories to address student misconceptions (Vosniadou & Skopeliti, 2014). Conceptual change denotes the process through which learners' existing (mis)conceptions develop into intended normative conceptions (Duit & Treagust, 2003; Vosniadou & Skopeliti, 2014). Conceptual change theories inform the process of modifying student misconceptions to help students establish normative understandings of the academic concepts to be learned (Vosniadou & Skopeliti, 2014).

After decades of research, two conflicting theoretical perspectives, revolutionary conceptual change and evolutionary conceptual change, have emerged (Abimbola, 1988; Özdemir & Clark, 2007; Taber, 2013). The revolutionary conceptual change perspective (also known as the knowledge-as-theory perspective) posits that learners' existing naïve knowledge is organized in a theory-like manner, and learners use their naïve theories to interpret and construct new concepts (Özdemir & Clark, 2007; Posner et al., 1982). Thus, learners' existing misconceptions are a potential barrier to new learning, and conceptual change is a revolutionary process that replaces learners' naïve theory-like knowledge structures with intended scientific conceptions. According to the revolutionary conceptual change perspective, successful instruction needs to help students confront their misconceptions, present the academic concept to students in a way that produces cognitive conflicts, and then help students abandon their misconceptions and adopt the new conceptions (Abimbola, 1988; Posner et al., 1982). Thus, the new conception needs to be intelligible, plausible, and fruitful (Posner et al., 1982). In other words, the new conception should be easy to understand, offer more explanatory power than the old conception, and demonstrate "the potential to be extended, to open up new areas of inquiry" (Posner et al., 1982, p. 214).

In contrast, the evolutionary conceptual change perspective (also known as knowledge-as-elements perspective) postulates that learners' prior naïve knowledge consists of relatively unstructured collections of quasi-independent elements (Abimbola, 1988; diSessa, 1993). From this viewpoint, conceptual change is an evolutionary process of correcting and enhancing existing knowledge elements and establishing and refining the relationships among conceptions. These naïve knowledge elements of students are called phenomenological primitives (p-prims) by diSessa (1993, 2014). P-prims are rooted in students' superficial interpretations of their daily experience. Students often use p-prims to interpret scientific phenomena and also their life experience. The key feature of p-prims is their primitiveness that enables the naïve knowledge elements to be self-explanatory. Unlike the revolutionary conceptual change perspective, researchers of evolutionary conceptual change believe that learners' existing (mis)conceptions should be considered as productive resources for constructing new concepts, and the purpose of instruction is to reconcile students' prior (mis)conceptions with new learning, rather than replacing them (Abimbola, 1988; diSessa, 2013, 2014). Although the composition of students' naïve knowledge elements can be productive when constructing new knowledge, few studies have investigated the productive aspects of students' prior knowledge (diSessa, 2014). Furthermore, according to the evolutionary conceptual change perspective, student misconceptions are contextually sensitive (diSessa, 2013; Özdemir & Clark, 2007). In different learning contexts, different knowledge elements of students may be activated and used in the knowledge construction process. When the learning contexts are similar, students may also form a variety of intermediate (mis)conceptions during conceptual change.

While debate between the two perspectives is ongoing (see diSessa, 2013 and Vosniadou, 2013), the current trend in conceptual change research has shown convergence (Vosniadou & Skopeliti, 2014). Vosniadou and Skopeliti (2014) proposed the framework theory approach, which tried to resolve the conflicts between the two perspectives. On the one hand, this approach agrees that conceptual change is an evolutionary process, naïve knowledge elements exist in students' prior knowledge, and student misconceptions are not independent of learning contexts. On the other hand, it argues that students tend to use coherent framework theories, which are generated from their naïve knowledge, to interpret phenomena. The framework theory approach posits that before students successfully understand the new academic concept, the interaction between the new concept and their existing framework theories results in various synthetic models, which are intermediate states of knowledge with partially correct interpretation (Vosniadou, 1994; Vosniadou & Skopeliti, 2014).

Nowadays, researchers of conceptual change theories share the ideas that (a) learners' own pre-instructional conceptions (also called naïve knowledge) are based on their daily experience; (b) learners' existing knowledge has an impact on the acquisition of new knowledge; and (c) student misconceptions are often entrenched and conceptual change is time consuming (Özdemir & Clark, 2007; Taber, 2013). Researchers agree that evolutionary conceptual change is a prerequisite of revolutionary conceptual change (Taber, 2013; Vosniadou & Skopeliti, 2014), and success in conceptual change requires tracking the development of learners' (mis)conceptions using real-time data of learning (diSessa, 2014; Vosniadou, 2013). With precise understanding of the nature and current status of student (mis)conceptions, instructors can choose proper strategies for accomplishing conceptual change, such as "directly challenging student conceptions," "ignoring them and simply teaching the canonical ideas," or "seeing learners' conceptions as useful (or necessary) starting points that need to be modified over time through a multistage conceptual trajectory" (Taber, 2014, p. 40).

Summary

Conceptual change theories stem from studies in the fields of science and mathematics education. However, they may also be valuable to studies in computer science education as they provide frameworks for understanding the formation and evolution of student misconceptions and offer instructional strategies for addressing student misconceptions. According to conceptual change theories, the modification of student misconceptions is an evolutionary process. Careful analysis of students' existing conceptions and considering learners' naïve knowledge as a productive resource for knowledge construction is vital to promote conceptual change (diSessa, 2014; Vosniadou & Skopeliti, 2014). The key to address student misconceptions is describing the knowledge acquisition process and tracking the evolution of learners' understandings using learner data (diSessa, 2013, 2014; Vosniadou, 2013). While conceptual change theories have been widely adopted to understand the development of student knowledge in math and science (Vosniadou & Skopeliti, 2014), they have received relatively little attention in CS education to date (Qian & Lehman, 2017; Sorva, 2012).

Feedback in Education

There is no doubt that feedback is important in education. However, researchers often define feedback differently and have not reached an agreement on its impact on learning. Studies about timing, complexity, and sources of feedback often report inconsistent results. This section reviews studies on the definitions of feedback in education, the effects of feedback, and the models for designing effective feedback.

Definitions of Feedback

Feedback is essential to help learners successfully construct new knowledge. Historically, researchers have defined feedback from three different perspectives: feedback as motivator, feedback as reinforcement, and feedback as information (Kulhavy & Wager, 1993). According to the motivational viewpoint, feedback is a motivator or incentive for enhancing learning performance. However, the mix of motivation and feedback makes it difficult to conceptualize how feedback works (Kulhavy & Wager, 1993). The feedback-as-reinforcement perspective posits that feedback producing a satisfying effect is likely to make the response repeated in the future. This idea was derived from E. L. Thorndike's Law of Effect and greatly developed by B. F. Skinners' study of programmed instruction (Kulhavy & Wager, 1993; Mory, 2004). These researchers believed that telling the learner his or her answer is correct would increase the probability of the learners making the same right response in the future. Hence, studies of the feedback-as-reinforcement perspective mainly focused on learners' correct responses and often ignored errors (Kulhavy, 1977).

Instead of concentrating on correct responses, the feedback-as-information perspective emphasizes learners' erroneous responses and considers feedback as information for correcting learners' errors and misunderstandings (Butler & Winne, 1995; Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Shute, 2008). Butler and Winne (1995) proposed that feedback is "information with which a learner can confirm, add to, overwrite, tune, or restructure information in memory, whether that information is domain knowledge, metacognitive knowledge, beliefs about self and tasks, or cognitive tactics and strategies" (p. 275). Hattie and Timperley (2007) defined feedback as information provided by an agent aiming to reduce the gap between the learner's current and intended understanding. According to Shute (2008), feedback is "information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning" (p. 154). While researchers nowadays do not deny that feedback in education may lead to changes in learners' motivation and reinforcement of learning, they agree that feedback in essence is information for facilitating learning (Hattie & Gan, 2011; Hattie & Timperley, 2007; Shute, 2008). More specifically, feedback is information provided by an agent to change learners' thinking or behavior for the purpose of enhancing learning (Hattie & Timperley, 2007; Shute, 2008).

Effects of Feedback

While feedback is treated differently by different researchers, overall it is powerful in learning and teaching within various learning contexts (Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Van der Kleij et al., 2015). However, previous studies have not reached an agreement on the effects of feedback (Kluger & DeNisi, 1996; Shute, 2008; Van der Kleij et al., 2015). Three major variables influencing the effects of feedback have been discussed by researchers: timing (immediate vs. delayed), complexity (simple vs. complex), and source (external vs. internal).

Timing of Feedback

Even though the timing of feedback has been widely studied, conflicting results exist in previous research (Kulik & Kulik, 1988; Shute, 2008; Van der Kleij et al., 2015). Results of some studies have favored immediate feedback, which refers to feedback provided immediately after the learner's response. In an early meta-analysis of findings on feedback timing, Kulik and Kulik (1988) concluded that immediate feedback was more effective than delayed feedback in actual classroom settings. According to the meta-analysis on effects of feedback in computer-based instruction conducted by Azevedo and Bernard (1995), studies using immediate posttests showed a mean weighted effect size of 0.80, while a mean effect size was estimated at 0.35 for studies involving delayed posttests. By studying students who learned Lisp programming using the ACT Programming Tutor, Corbett and Anderson (2001) noted that immediate feedback increased the learning rate while producing the equivalent performance. In their study, three versions of feedback were offered to students: immediate feedback with automatic error correction, immediate feedback with learner-controlled error correction, and delayed feedback on learners' demand with learner-controlled error correction. Students in the first group received immediate feedback when they made a mistake, and the tutor automatically corrected the errors for them. The second group students also got immediate feedback on bugs but had to fix errors by themselves. The third group students only saw feedback when they requested. Though the three groups performed equivalently on the tests, the results indicated that immediate-feedback groups learned the content faster than the demand-feedback group (Corbett & Anderson, 2001).

Delayed feedback means that an interval exists between response and feedback. The interval can be minutes, hours, days, or longer in different studies (Shute, 2008). Previous studies (Butler, Karpicke, & Roediger, 2007; Kulik & Kulik, 1988) reported the superiority of delayed feedback in testing situations. In an experiment investigating the feedback effects of different degrees of delay from 0 s to 30 s, Schroth (1992) found that delayed feedback was beneficial to learning transfer while it decreased the rate of initial learning. Kulhavy (1977) pointed out that with delayed feedback learners showed better performance in retention tests because of the delay-retention effect (Kulhavy & Anderson, 1972). The delay-retention effect posits that corrective information will be more effective when it is delayed because learners may forget the initial errors during the interval. On the contrary, another study about feedback timing and retention indicated that delayed feedback led to greater retention of initial lesson responses than immediate feedback, whether the initial responses were right or wrong (Clariana et al., 2000). Overall, it seems that delayed feedback is superior to immediate feedback in some circumstances.

In addition to the conflicting reports of effectiveness of immediate and delayed feedback, some other studies also reported that the timing of feedback showed no significant effects on learning (Jaehnig & Miller, 2007; Mory, 2004). However, it is not wise to discuss the effects of feedback simply considering its timing. First, even though immediate feedback is easy to define, the meaning of delayed feedback is often ambiguous (Van der Kleij et al., 2015). Does a one-minute delay have the same effects as a one-hour delay? It is difficult to answer. Furthermore, with different intended learning outcomes, immediate and delayed feedback may show different effects (Shute, 2008). Immediate feedback seems to be more beneficial to lower order learning outcomes while delayed feedback appears to be more effective on higher order learning outcomes (Van der Kleij et al., 2015). Therefore, without discussing the content of feedback and the learning contexts, directly comparing the effects of immediate and delayed feedback probably is meaningless.

Complexity of Feedback

Another commonly researched variable that may contribute to the effects of feedback is complexity (Mory, 2004; Shute, 2008; Van der Kleij et al., 2015). When discussing the complexity of feedback, two similar categorizations are often applied. The first one categorizes feedback as outcome feedback and cognitive feedback (Balzer, Doherty, & O'Connor, 1989; Butler & Winne, 1995). Outcome feedback, sometimes also called "knowledge of results," provides verification information of the correctness of a learner response. Rather than merely telling a learner whether his or her response is correct or incorrect, cognitive feedback conveys additional information for improving the response. The complexity of cognitive feedback may vary depending on how much extra information it includes.

Another categorization contains three types of feedback: knowledge of results (KR), knowledge of correct response (KCR), and elaborated feedback (EF) (Dempsey, Driscoll, & Swindell, 1993; Van der Kleij et al., 2015). KR is same as the outcome feedback in the former categorization and tells the learner whether his or her answer is right or wrong. KCR provides the correct answer for learners. EF contains additional corrective information such as hints and extra learning material in order to guide the learner towards the right response. Though different terms are used, the two categorizations are similar to each other. Outcome feedback is the same as KR. As KCR and EF provide corrective information rather than only verification information, they both can be considered as cognitive feedback. Hence, it is possible to combine the two categorizations.

According to Kulhavy and Stock (1989), effective feedback usually consists of two essential and separable components: verification and elaboration. Therefore, we can merge the two categorizations in this way: if the feedback only offers verification, it is outcome feedback or KR; if the feedback contains additional corrective information, such as the correct answer, clarification of the correct answer, explanation of the errors, guidance for revision, and so forth, which are aimed at correcting the learner's misunderstanding, it is cognitive feedback or elaborated feedback. Thus, KCR, simply providing the correct answer, can be considered as a special version of elaborated feedback.

Obviously, cognitive feedback is more complex than outcome feedback. Does the complexity of feedback matter? Previous research has shown inconsistent results (Mory, 2004). In a study using five modes of feedback: no feedback, outcome feedback, KCR, cognitive feedback, and the combination of outcome feedback and cognitive feedback, Gilman (1969) reported that in general cognitive feedback was more effective than outcome feedback, and the combination group showed better immediate retention. On the contrary, Wentling (1973) compared using outcome feedback and KCR for male high school students in the course General Automobile Mechanics and found that outcome feedback resulted in better immediate achievement in unit tests. In the dissertation of Lee (1985), no significant differences were found between using outcome and cognitive feedback.

Specifically, regarding cognitive feedback, various studies have been conducted to compare the effectiveness of different types of elaborative information (Jaehnig & Miller, 2007; Sleeman, Kelly, Martinak, Ward, & Moore, 1989; Van der Kleij et al., 2015). For instance, in a study of high school algebra learners, Sleeman et al. (1989) applied two different approaches of providing feedback: model-based remediation (MBR) and reteaching. MBR offers "procedurally orientated remediation of specific errors found in a student's solutions before reteaching a correct strategy" (Sleeman et al., 1989, p. 552). Reteaching refers to simply teaching the correct method again without addressing learners' errors. While both approaches were more effective than no feedback, Sleeman et al. (1989) noted that MBR was not superior to reteaching.

By systematically reviewing different types of feedback in programmed instruction, Jaehnig and Miller (2007) indicated that though more time is required for instructional design and learning, elaborated feedback is more effective than simply providing the correct response (KCR). According to a meta-analysis of feedback effects in computer-based learning environments (Van der Kleij et al., 2015), cognitive feedback with an explanation or other additional information for modifying learners' misunderstanding (effect size 0.49) is more effective than cognitive feedback merely providing the correct answer (effect size 0.32) and both types of cognitive feedback are better than outcome feedback (effect size 0.05). Van der Kleij et al. (2015) also argued that elaborated cognitive feedback was more effective for higher order learning outcomes than KCR and outcome feedback.

While many studies have discussed the effects of feedback of different complexities, researchers have not reached a consensus. Apparently, simply comparing the effectiveness of outcome feedback and cognitive feedback without considering learner characteristics and learning contexts is not sensible. Fifth graders who learn English vocabulary and college students who learn Java programming probably need feedback of distinct complexities. Furthermore, cognitive feedback is difficult to define as the elaboration can be varied in different studies. As a common type of cognitive feedback, providing the correct response (KCR) seems to be consistently defined by different studies. However, offering the correct answer to a multiple-choice question (e.g., Kulhavy & Stock, 1989) compared to a programming problem (e.g., Corbett & Anderson, 2001) may lead to different learning outcomes. How cognitive feedback works really depends on how much information it contains and how the information is presented. Thus, it is crucial to discuss and compare the influence of feedback's complexity within a specific design and setting. However, researchers have reached agreement that providing feedback is better than no feedback for enhancing learning, and generally cognitive feedback seems superior to outcome feedback (Balzer et al., 1989; Butler & Winne, 1995; Hattie & Timperley, 2007; Van der Kleij et al., 2015).

Source of Feedback

Feedback can also be categorized as external and internal according to its sources. Traditionally, researchers have focused on feedback offered to students by an external source, such as a human or a computer (Bangert-Drowns et al., 1991; Corbett & Anderson, 2001; Van der Kleij et al., 2015). External feedback from teachers on tests or in a classroom setting is frequently discussed by researchers (Hattie & Timperley, 2007; Kulhavy & Stock, 1989) because teachers are usually the primary source of feedback. However, a new strand of research on feedback of peers has emerged (Lu & Law, 2012). As the popularity of formative assessment has grown rapidly, peer assessment has been considered as a potential way to effectively provide formative assessment (Gielen, Peeters, Dochy, Onghena, & Struyven, 2010; Sadler, 1989). Peer assessment, which refers to evaluating peers' work and giving constructive feedback, consists of two components: peer grading and peer feedback (Lu & Law, 2012). Peer grading is the process in which assessors assign grades to peers' work by applying criteria and standards; peer feedback refers to providing constructive comments on peers' work. We may consider peer grading as outcome feedback from peers and peer feedback as cognitive feedback from peers. Though it is regarded as a reliable approach, peer grading alone is less effective than peer grading plus peer feedback (Liu & Carless, 2006; Lu & Law, 2012). In addition, peer feedback benefits both assessors and assessees (Gielen et al., 2010; Lu & Law, 2012). However, providing peer feedback seems to be more beneficial for enhancing understandings and improving learning than simply receiving peer feedback (Li, Liu, & Steckelberg, 2010; Patchan & Schunn, 2015). Perhaps, this is because that students do not always take actions on implementing feedback (Nelson & Schunn, 2009), or they prefer adopting teacher feedback to accepting peer feedback (Yang, Badger, & Yu, 2006).

In addition to humans, with the development of technology, computers have become another important source of feedback for learners. Since the 1960s, researchers have strived to build computer-based systems that can automatically provide effective feedback for learners (Anderson, Boyle, & Reiser, 1985; Smith & Sherwood, 1976; Ulloa, 1980). In these systems, students "receive essentially instantaneous reinforcement of correct work and assistance where they are having difficulty" (Smith & Sherwood, 1976, p. 334). There are two major types of this kind of learning system. The first type is usually called Computer Aided Instruction (CAI) or Computer-Based Instruction (CBI) and offers immediate feedback to learners based on their answers (VanLehn, 2011). The second type is often called Intelligent Tutoring System (ITS) and is "characterized by giving students an electronic form, natural language dialogue, simulated instrument panel, or other user interface that allows them to enter the steps required for solving the problem" (VanLehn, 2011, p. 198). The major difference between them is that CAI usually provides feedback such as a hint or a congratulatory message for the learner without scrutinizing the details of the response and the process of constructing the response, while an ITS typically offers concrete step-based feedback either during the problem-solving process or after the solution is submitted with specific supportive information based on the learner's errors and (mis)understandings.

Although some studies found that feedback offered by humans sometimes is more flexible and effective than that from a machine (Merrill, Reiser, Ranney, & Trafton, 1992; VanLehn et al., 2007), there are also researchers indicating that computer-provided feedback is as effective as human-provided feedback (VanLehn, 2011) and sometimes even leads to better learning rates and performance (Anderson et al., 1985). By comparing human tutors and ITSs, Merrill et al. (1992) noted that the effectiveness was similar when the computer systems could provide as much assistance as necessary like human tutors. As digital learning systems nowadays can capture a tremendous amount of learner data, applying data-driven techniques to give automated feedback to learners has become more feasible (Gerdes et al., 2017; Rivers & Koedinger, 2017). By analyzing errors a specific learner makes, data-driven learning systems are able to offer customized feedback to the learner in order to help him or her correct errors and change misconceptions.

In contrast to external feedback, internal feedback is generated by the learner and is crucial to self-regulation that “guides cognitive activities during which knowledge is accreted, tuned, and restructured” (Butler & Winne, 1995, p. 246). Hence, as an essential component of self-regulated learning, learners use internal feedback to set learning goals, monitor their progress, judge their performance relative to the goals, and act to reduce the discrepancies between goals and outcomes (Nicol & Macfarlane-Dick, 2006). Sadler (1989) argued that it is important to develop learners’ ability to assess their own work, appreciate high quality work, and generate internal feedback for closing the gap. As it is ubiquitous during the learning process, internal feedback is vital to the effectiveness of external feedback. A learner may generate internal feedback when he or she is working on a task or after receiving external feedback about his or her response. When external feedback such as outcome feedback provides minimal information about how to self-regulate, it may not lead to effective internal feedback and improved performance (Butler & Winne, 1995; Moos, 2011). Cognitive feedback, in contrast to outcome feedback, usually gives learners information that guides cognitive activities for locating and fixing the errors. However, no matter whether it is outcome or cognitive feedback, “students filter information provided by external feedback through knowledge and beliefs, applying conditional knowledge to identify cues” (Butler & Winne, 1995, p. 264). Hence, the effectiveness of the external feedback actually depends on the learner’s interpretation of the information based on their prior knowledge and beliefs, rather than the information itself.

In general, internal and external feedback work together to improve learning performance and close the gap between the current and intended understandings (Merrill et al., 1992). In terms of feedback, learners should be considered as “having a proactive rather than a reactive role in generating and using feedback” (Nicol & Macfarlane-Dick, 2006, p. 199). Therefore, when we design and provide feedback for learners, it is important to deliberate how external feedback will influence internal feedback and guide self-regulation.

Models of Feedback

As a number of factors may influence the effect of feedback on learning, including timing, complexity, and sources of feedback (Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Van der Kleij et al., 2015), researchers have developed different models of feedback to explain how feedback facilitates learning and provide guidelines for designing effective feedback. Well-known models of feedback include the certitude model with a focus on learner response confidence (Kulhavy & Stock, 1989), the five-stage model emphasizing learners' mindful reflection (Bangert-Drowns et al., 1991), and the connectionist model concentrating on the retention of initial lesson responses (Clariana et al., 2000). Each of these models mainly addresses one factor that may contribute to feedback's effects, such as confidence (Kulhavy & Stock, 1989), self-regulation (Bangert-Drowns et al., 1991; Butler & Winne, 1995), and timing (Clariana et al., 2000).

Recently, a more inclusive model, the visibility model of feedback, with an emphasis on visualizing learners' current knowledge states, has been developed and become widely accepted (Hattie & Gan, 2011; Hattie & Timperley, 2007). Hattie and Timperley (2007) indicated that the previous commonly debated issues about feedback such as timing and complexity were mainly due to the lack of recognition of the various feedback levels. According to the visibility model, feedback is information for reducing "the discrepancy between what is understood and what is aimed to be understood" and is the most powerful when it makes learning visible to both the teacher and the learner (Hattie & Gan, 2011, p. 257-258). *Visibility* means that effective feedback needs to answer three major questions: "Where am I going?," "How am I going?," and "Where to next?" (Hattie & Timperley, 2007). In other words, procedures for designing effective feedback include (1) clearly describing the desired learning outcomes, (2) precisely analyzing learners' current knowledge states, and (3) identifying the discrepancy between the current states and the intended outcomes and providing information for reducing the discrepancy and enhancing learning (Hattie & Gan, 2011; Hattie & Timperley, 2007).

Feedback answering the three questions works at four different levels: task level, process level, self-regulation level, and self-level (Hattie & Gan, 2011; Hattie & Timperley, 2007). The task-level feedback provides information about the task or product, such as the learner's performance on the task and additional task-related information (e.g., "Your answer is almost correct, but X in the problem is 3 not 5."). It is commonly used in the classroom and usually is not generalizable to other tasks (Hattie & Gan, 2011). While providing task-related information typically is only effective for building surface knowledge, the acquisition of correct task-related information is “a pedestal on which the processing and self-regulation is effectively built” (Hattie & Timperley, 2007, p. 91).

Feedback aiming at the process level provides “task processing strategies and cues for information search” (Hattie & Gan, 2011, p. 260). For instance, a computer science teacher may tell a student, “Your program is working, but you did not use the correct variable types.” The teacher’s feedback offers cues for debugging the program to the student. Thus, feedback at process level is important for learners to detect errors and develop correct understandings. Though process-level feedback is powerful for enhancing deeper learning, it often interacts with task-level feedback as task information is vital to process the task (Hattie & Gan, 2011).

Feedback at the self-regulation level aims at directing learners’ self-evaluation, boosting learners’ self-efficacy, increasing learners’ effort in task engagement, and improving other self-regulated activities (e.g., “Your program is correct, but how can you revise it to improve its performance?”). It mainly tries to help learners develop skills of monitoring the learning process, evaluating the information provided, and reflecting on the learning outcomes (Hattie & Gan, 2011).

Self-level feedback is directed to the “self” without providing information about how to enhance the task performance or improve the product (e.g., “Well done”, “Good job”). As little task-related information is contained in such praise, only using praise often has little impact on achievement (Hattie & Timperley, 2007; Kluger & DeNisi, 1996).

In summary, the core of the visibility model of feedback is to visualize the discrepancy between learners’ current understanding and the desired understanding. Hence, to design effective feedback, it is essential to clearly describe the learning goals and precisely understand the learner’s current progress, and then present information for closing the gap. Furthermore, the discrepancy should be visible for both teachers and learners (Hattie & Gan, 2011). For teachers, they need to know what challenges and difficulties students face so that they can provide appropriate feedback. For learners, they need to know what errors they have made and what the desired outcome is so that they may receive the feedback effectively and take actions to change their (mis)understandings and improve their responses. Finally, the visibility model proposes four levels of feedback and emphasizes that effective feedback needs to be designed and provided at the appropriate operational level(s) (Hattie & Gan, 2011). Hence, for different learners (e.g., novices, experts) and different learning settings, feedback needs to include different levels of information (task, process, self-regulation, self, or combined).

Summary

Although historically researchers treated educational feedback as motivator or reinforcement of learning, nowadays they agree that feedback is information for facilitating learning (Hattie & Gan, 2011; Hattie & Timperley, 2007; Shute, 2008). Admittedly, feedback may lead to an increase in motivation or reinforcement of learning as consequences. In essence, it is information provided by an agent to change learners' thinking or behavior for the purpose of enhancing learning (Hattie & Timperley, 2007; Shute, 2008). A variety of factors such as timing, complexity, and sources may influence the effects of feedback within different learning contexts (Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Van der Kleij et al., 2015). According to the visibility model, the problem of traditional feedback design is that it neglects to examine learners' current (mis)conceptions but simply provides corrective information for fixing superficial learner errors (Hattie & Gan, 2011). The new model of feedback requires scrutinizing learners' erroneous responses, to grasp their positions on the trajectory towards the success of the learning goals, and provide corrective information targeted at addressing student misconceptions (Hattie & Gan, 2011; Hattie & Timperley, 2007).

Student Misconceptions in Introductory Programming

In the learning of programming, student misconceptions are students' deficient or erroneous understandings of programming concepts (Qian & Lehman, 2017; Sorva, 2013; Taber, 2013). In previous literature, a variety of terms have been used to describe students' inaccurate understandings in learning to program, such as "misconceptions" (Sorva, 2013), "difficulties" (du Boulay, 1986), "errors" (Sleeman et al., 1986), "bugs" (Pea, 1986), "mistakes" (Altadmri & Brown, 2015), and so forth. With these different terms, researchers have discussed students' syntax errors in the code, misunderstandings of programming concepts, difficulties in writing and debugging programs, and so on (Sorva, 2013). While various student misunderstandings and errors are often lumped together as "misconceptions," qualitative differences exist between a simple syntax error in a loop statement, conceptual misunderstandings of loops, and challenges of using loop constructs to solve problems. However, these difficulties are related to each other, and problems of students' conceptual understandings are the pivot that may lead to syntactic errors, logic errors, and other difficulties (Bayman & Mayer, 1988; de Raadt, 2008; Ebrahimi 1994; Lopez et al., 2008; Qian & Lehman, 2017). In this section, studies about student misconceptions and related difficulties in introductory programming are reviewed and organized by two themes including understanding and addressing student misconceptions.

Understanding Student Misconceptions

Introductory CS courses are difficult for beginners (Guzdial, 2015; McCracken et al., 2001), and students often exhibit misconceptions that impede their learning of introductory programming (Altadmri & Brown, 2015; Sorva, 2013). Qian and Lehman (2017) summarized literature regarding common misconceptions and other difficulties in introductory programming. For instance, variables are a very basic concept in most of the programming languages, but novices may mistakenly believe that the computer understands variables by the English meanings of their names, even though variable names are arbitrary (Kaczmarczyk et al., 2010; Sleeman et al., 1986). Sequential execution of code is another challenging concept for beginners (du Boulay, 1986; Simon, 2011). For instance, students may mistakenly believe that when the Boolean expression of a conditional statement becomes true, even if this occurs twenty lines below the conditional statement, the program will go back and execute the code in that conditional block (Pea, 1986). High-level concepts such as classes, objects, instances, and their relationships in object-oriented programming (OOP) also often confuse students (Guzdial, 1995; Holland et al., 1997; Ragonis & Ben-Ari, 2005; Sorva, 2013). Ragonis and Ben-Ari (2005) conducted a two-year long study and identified 58 conceptions and difficulties students encountered in a high school introductory Java programming course.

Students in introductory programming courses may also exhibit syntax errors when writing programs to solve problems. Researchers have cataloged common syntax errors of students in introductory programming (Altadmri & Brown, 2015; Hristova, Misra, Rutter, & Mercuri, 2003; Jackson et al., 2005; Sirkia & Sorva, 2012). After analyzing students' compilation errors in a freshman Java course, Jackson et al. (2005) reported that the top three errors students made were forgetting variable declaration, missing semicolons, and using illegal start of expressions. In Java programming, a variable has to be declared before being used, but students often forget to declare variables. A semicolon is a required punctuation mark to end a statement in Java, but beginners often forget to add it. The third error, using illegal start of expression, usually results from incorrect construction of Java expressions, such as using wrong punctuation marks in expressions. By analyzing millions of errors of students who learned Java programming using the BlueJ IDE, Altadmri and Brown (2015) provided a list of 18 common mistakes in Java programming and noted that mismatching parentheses, brackets, or quotation marks is the most common syntactic error. Another common novice mistake is incorrectly using the assignment operator (=) instead of the comparison operator (==) (e.g., *if (a = b)*) (Hristova et al., 2003; Sirkia & Sorva, 2012).

In addition, beginners usually lack well-established programming strategies (Clancy & Linn, 1999; Davies, 1993; Lister et al., 2006; Sajaniemi & Prieto, 2005; Soloway 1986) and then face difficulties with planning, composing, and debugging programs. Strategic problems include failing to understand and decompose the task (Muller, 2005; Robins et al., 2006), forgetting to test boundary conditions and unexpected cases (Fisler et al. 2016; Sajaniemi & Kuittinen, 2005; Spohrer & Soloway, 1986), and inappropriately tracing their code and locating errors (Ben-David Kolikant & Mussai, 2008; Fitzgerald et al., 2008; McCauley et al., 2008).

Qian and Lehman (2017) described factors that may contribute to students' misconceptions in learning to program. Major factors that contribute to students' misconceptions include interference caused by prior knowledge (Clancy, 2004; Miller, 2014) and flawed mental models of computer operation (Guzdial, 2015; Sorva, 2013). Novice students may mistakenly use concepts they learned in math to try to understand programming concepts (e.g., variables), which look similar but mean something quite different (Clancy, 2004; Qian & Lehman, 2017). As most programming languages are natural-language-based, students' existing knowledge of natural language may hinder their construction of the meanings of programming concepts (Bruckman & Edwards, 1999; du Boulay, 1986; Miller 2014). In addition, unlike experts, beginners' conceptual knowledge is often fragmentary and not organized into meaningful patterns (Clancy & Linn, 1999; Lister, 2011; McCauley et al., 2008; Sajaniemi & Prieto, 2005; Whalley et al., 2006). Thus, they may only be able to understand programs in a line-by-line manner and then fail to holistically evaluate and properly debug a program (Ben-David Kolikant & Mussai, 2008; Lister et al., 2006). Students in introductory programming courses also often hold flawed mental models of the notional machine, which refers to an abstract computer that executes code in the programmer's mind (du Boulay, 1986; Guzdial, 2015; Sorva, 2013).

Without correct understanding of the notional machine, a student may fail to understand the sequential execution of statements (du Boulay, 1986; Simon, 2011).

Addressing Student Misconceptions

In computer science education, researchers and educators have developed various instructional approaches and tools to address students' misconceptions in introductory programming.

Instructional Approaches

Using program examples in instruction is one effective approach to address students' understanding. Previous research revealed that using worked-out examples in instruction can improve students' performance in solving problems (Ginat et al., 2011). Asking students to comprehend and explain example programs can help to disclose students' misconceptions and develop their program composing and debugging skills (McCauley et al., 2008; Teague & Lister, 2014; Vainio & Sajaniemi, 2007). Another approach is explicitly teaching programming strategies in introductory programming. de Raadt (2008) reported that after students received explicit instruction in programming strategies, they showed improvements in overall programming performance and better ability to apply programming strategies to solve problems. Muller et al. (2007) found that pattern-oriented instruction (POI) can help to reduce students' cognitive load in programming and improve their ability to decompose problems and construct solutions.

Other approaches include using a concept inventory (CI) (Goldman et al., 2010; Taylor et al., 2014; Tew, 2010) and Peer Instruction (PI) (Porter et al., 2013). A concept inventory is an assessment that aims to evaluate students' understanding of a group of concepts (Goldman et al., 2010; Tew, 2010). Using a concept inventory to evaluate students' understanding of key programming concepts enables instructors to identify common misconceptions students have and then improve their instruction based on the misconceptions (Taylor et al., 2014). Another instructional approach is Peer Instruction, which focuses on engaging students in active learning of new concepts. It includes three steps: answering a question individually, having discussions with peers, and reconsidering the question again (Simon et al., 2010). Previous research on Peer

Instruction indicated that Peer Instruction can effectively improve students' learning performance in introductory programming (Porter et al., 2013; Simon et al., 2010).

Instructional Tools

In computer science education, many programming environments and tools have been developed to address students' misconceptions in introductory programming. Three major types of the instructional tools are novice programming environments, code visualization tools, and automated assessment systems.

Novice Programming Environments

Block-based programming environments such as Scratch (Resnick et al., 2009) and Alice (Dann, Cosgrove, Slater, Culyba, & Cooper, 2012) can prevent syntax errors and help novices develop a better understanding of programming concepts (Price & Barnes, 2015; Weintrop & Wilensky, 2015). Natural-language-like programming languages have also been developed to reduce learners' programming errors and enhance learning performance (Bruckman & Edwards, 1999). By embedding Whyline, a special designed debugging interface for novice programmers, Ko and Myers (2005) reported significant improvements in students' debugging skills.

Code Visualization Tools

Code visualization tools are tools that can illustrate the process of code execution and variable states (Sorva et al., 2013). One well-known code visualization tool is Python Tutor (Guo, 2013), which was originally designed for visualizing the execution of Python code step by step and now supports other programming languages such as Java, JavaScript, TypeScript, and so forth. Another example is Greenfoot, which focuses on visualizing OOP concepts to help students better understand Java classes and objects and the execution flow of Java programs (Kölling, 2010). As beginners often hold problematic mental models of the notional machine, Sorva (2012) developed UUhistle that focuses on the visualization of the notional machine for Python programming. Because one important source of student misconceptions in introductory programming is their misunderstandings of code execution and problematic mental models of the computer system, code visualization tools have been helpful to support addressing certain student misconceptions (Sirkia & Sorva, 2012; Sorva et al., 2013). On the other hand, instructors should not assume code visualization tools can benefit all students in all teaching contexts, because these tools may also increase students' cognitive load during learning (Guzdial, 2015; Sorva, 2012; Sorva et al., 2013).

Automated Assessment Systems

Automated assessment systems have also been widely used in introductory programming classes to support teaching and learning (Douce et al., 2005; Pettit, Homer, & Gee, 2017). An automated assessment system is a tool that can automatically evaluate the correctness of students' programs and provide immediate feedback (De-La-Fuente-Valentín et al., 2013; Gerdes et al., 2017). With the student data, especially students' erroneous programs, two types of feedback systems have been developed and integrated into automated assessment systems.

The first type of feedback system uses artificial intelligence (AI) techniques to analyze students' programs and generate personalized feedback for students (Barnes & Stamper, 2010; Rivers & Koedinger, 2017; Xu & Chee, 2003). With such an intelligent feedback component, an automated assessment system becomes an intelligent tutoring system that can not only grade students' programs but also provide automated feedback. iSnap is an intelligent tutoring system that can automatically generate hints for Snap programming learners (Price, Dong, & Lipovac, 2017). Price et al. (2017) reported that hints generated by iSnap were helpful to address simple problems in students' code. While such systems seem to be an ideal solution to help teachers identify and address student misconceptions, they are not mature yet and can only handle simple programs.

The other type of feedback system uses manually designed feedback messages for common student errors identified using the student data in the automated assessment system (Becker, 2016; Denny, Luxton-Reilly, & Carpenter, 2014; Pettit et al., 2017). Decaf is such a system (Becker, 2016). In a study of using Decaf to teach a Java-based CS1 class, Becker (2016) first used student data in the automated assessment system to identify 30 common compilation errors and then designed feedback by enhancing the raw Java error messages. His results showed that the 30 compilation errors accounted for 78% of all errors, and the group receiving feedback messages made 32% fewer errors than the group only seeing the raw Java compiler error messages. Prior studies on automated assessment systems with such feedback components have two issues. First, previous studies using this type of feedback component have only focused on students' compilation errors (Pettit et al., 2017). Second, the effectiveness of using enhanced compiler error messages as feedback is still questionable (Denny et al., 2014; Pettit et al., 2017).

Summary

Student misconceptions can interfere with learning of programming, and a variety of factors may contribute to these inaccurate understandings (Clancy, 2004; Qian & Lehman, 2017; Smith et al., 1994). While previous studies have cataloged a broad range of student misconceptions including syntax errors and other difficulties caused by misconceptions, most of them have focused on post-secondary students (e.g., Altadmri & Brown, 2015; Hristova et al., 2003; Jackson et al., 2005; Sirkia & Sorva, 2012). In addition, researchers and educators have developed various instructional approaches and tools to address students' misconceptions in introductory programming. Of particular interest for this study is the development of automated assessment systems that can automatically assess students' programs and provide immediate feedback to help students learn (Douce et al., 2005; Gerdes et al., 2017). While many automated assessment systems have been developed and tested by researchers, most systems to date either provide feedback based on compiler errors (e.g., Becker, 2016) or provide direct feedback for correcting simple errors in code (e.g., Gerdes et al., 2017).

Contribution of this Study

This study implemented a data-driven approach to identify secondary school students' misconceptions in introductory programming using both their compilation and test errors and provide targeted feedback to promote students' conceptual change. While previous studies have investigated a broad range of student misconceptions, most of them have focused on post-secondary students. As CS education has been expanding into K-12 schools, more information is needed to understand misconceptions among pre-college learners such as high school students who take introductory programming courses. Second, although previous studies in computer science education have discussed student misconceptions from a variety of perspectives, little work has drawn on our understanding of evolutionary conceptual change and appropriate use of data-driven feedback to promote conceptual change. Finally, while many automated assessment systems have been developed and tested by researchers, most systems to date either provide feedback based on compiler errors (e.g., Becker, 2016) or provide direct feedback for correcting simple errors in code (e.g., Gerdes et al., 2017). This study focused on analyzing both compilation and test errors in students' programs to understand and address student misconceptions.

CHAPTER 3: METHODOLOGY

The purpose of this study was to examine secondary school students' common misconceptions in introductory programming and investigate how feedback affected the evolution of students' (mis)conceptions using a data-driven approach. A two-stage exploratory design-based study was implemented. In the first stage, common misconceptions exhibited by students in solving programming problems using an automated learning system were identified. In the second stage, targeted feedback designed to address identified misconceptions was integrated into the automated learning system, and the effects of the use of that feedback with a new group of students were assessed. This section introduces the overarching methodological framework, settings and participants, and research procedures of conducting the study.

Methodological Framework: Design-Based Research (DBR)

This study used design-based research (DBR) (Anderson & Shattuck, 2012) as the overarching methodological framework. DBR is a methodology that guides the design, implementation, evaluation, and refinement of interventions to complex educational problems in real educational contexts (Anderson & Shattuck, 2012; Brown, 1992; McKenney & Reeves, 2014). DBR studies seek to simultaneously solve real-world problems in classroom settings and develop principles or theories for helping others facing similar situations (Anderson & Shattuck, 2012; McKenney & Reeves, 2014). The iterative process of conducting a typical DBR study includes analyzing problems, designing solutions, evaluating solutions, and reflecting on the results (McKenney & Reeves, 2014).

This study was aimed at solving a complex practical problem in classroom teaching and contributing to a theory of learning at the same time. DBR is a methodology that guides such studies (Brown, 1992; McKenney & Reeves, 2014). Understanding and addressing student misconceptions is complicated but important to classroom teachers. However, innovative interventions from traditional controlled laboratory settings often face challenges when transferred to real world classrooms (Brown, 1992). Therefore, this study adopted DBR as the methodological framework, which suggests iteratively designing, testing, and improving the intricate solution in classroom settings.

Settings and Participants

The research subjects in this study were two groups of high school students enrolled in two sections of a Java-based programming course as part of a 2017 summer residential program for gifted and talented students.

The Summer Residential Program and Courses

The setting for this study was a summer residential program that has been offered by Gifted Education Resource Institute (GERI) at Purdue University for more than four decades. The goal of this program is to help gifted and talented students from across the country and around the world develop their talents and expand their abilities. The 2017 summer program was held from July 2 to July 29, 2017 and consisted of two two-week sections. Section 1 was from July 2 to July 15, 2017. Section 2 was from July 16 to July 29, 2017. The fee to attend one section of the summer residential program was $2,400.

Participating students could choose to attend only one section or both. During a given section of the summer program, each student could select a morning class and an afternoon class based on his or her grade level. Three levels of classes for students of different grades were offered, including Comet classes for students who had completed grade 5 or 6, Star classes for students who had completed grade 7 or 8, and Pulsar classes for students who had completed grade 9, 10, 11, or 12. At the end of each class in a section, instead of receiving a grade, every student received a general evaluation of his or her thinking skills, social skills, and self-regulation exhibited in class.

The introductory Java-based programming class in this study was called *Programming and Computational Thinking*, and was offered to Pulsar students in both sections of the summer residential program (Pulsar 1 and Pulsar 2) in the morning from 8:30 to 11:30 every weekday. The researcher was the instructor of the introductory Java programming course. The major topics covered in this course were Program Structure, Input/Output (I/O), Variables and Operators, Conditionals, and Loops. Appendix A presents the course syllabus. The IDE (integrated development environment) used in the class was DrJava (version: drjava-20160913-225446). The JDK (Java SE Development Kit) version was JDK 8.

Typically, during every class session, the instructor started with a 30-minute lecture to review previously learned content (e.g., Input and Output methods in Java) and introduce new course content (e.g., the syntax of using Conditionals). After the lecture, the instructor used worked-out examples to show how to solve problems with the programming statements students had learned about. When the demonstration was done, students had about an hour to solve problems individually using the automated assessment system Mulberry. After the topic of Loops was introduced, students started individual and team projects based on their choices (e.g., design a text-based interactive game).

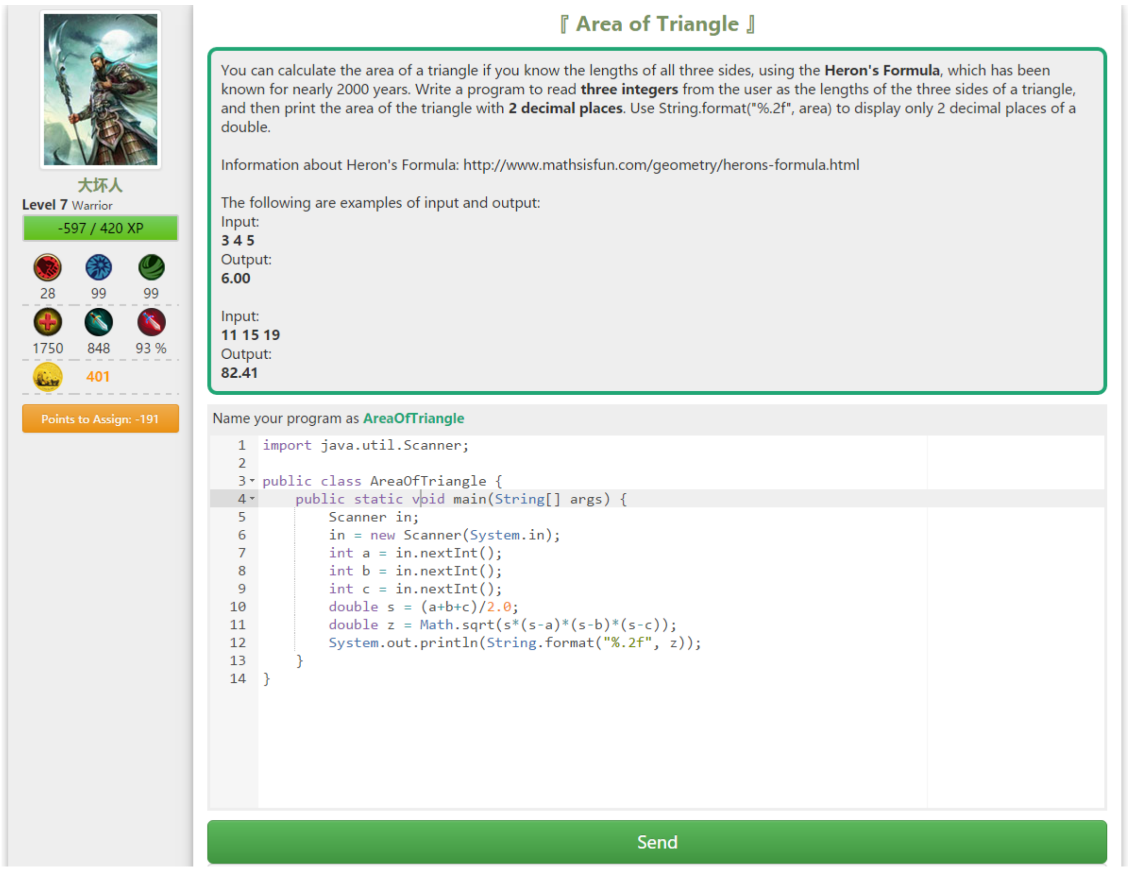
Participants

Participants in this study were two groups of high school (Pulsar) students, a total of 25, who took *Programming and Computational Thinking* in two different sections of the summer residential program. The student recruitment was conducted by GERI. To be accepted by this summer residential program, students had to be identified as high ability according to the GERI criteria. First, students completed an application form and wrote a statement of purpose explaining their desire and motivation to participate this program. Second, students were required to provide two of the following documents to demonstrate their talents: “a) a transcript showing a GPA of 3.5/4.0 in the talent area; b) an intelligence test report with a minimum score of 120; c) national achievement or aptitude test results at or above the 90th percentile in a specific area of study; d) a recommendation letter from a teacher or mentor in the talent area; e) documentation of involvement in the talent area” (GERI Website, 2018).

Group 1 of this study (Pulsar 1) originally had 15 students, and group 2 (Pulsar 2) had 10 students. However, one student of group 1 was found to have cheated when solving problems, so that the student’s problem solutions were not an accurate measure of performance. Another student of group 1 was the champion of a programming competition in his hometown who solved all the problems in Mulberry within two days and so was considered an outlier in terms of knowledge and ability. Therefore, these two students were not considered as participants of this study and were excluded from the data analysis. In the end, the participants of this study were 13 students (9 boys and 4 girls) in group 1 (Pulsar 1) and 10 students (7 boys and 3 girls) in group 2 (Pulsar 2).

Mulberry System

Mulberry is a programming learning system designed for Java learners and developed by the author. It has a pool of 51 programming problems, and students are required to write short programs to produce the correct output to solve the problems. Every problem has several test cases, which are pairs of input data and expected output. Mulberry automatically assesses students' solutions to each problem by using test cases and comparing the output of their programs with the expected output. A student solution is considered as correct when its output matches the expected output for all the test cases. When a student submits a program producing the incorrect output, he or she receives immediate feedback from the system and can try multiple times until his or her solution is correct. Mulberry collects all the programs from students when they attempt to solve the problems. Figure 3.1 shows the major user interface (UI) of Mulberry where students can read the problem description and submit the solution. The development of feedback for the system to address students' misconceptions is described later in this chapter.



『 Area of Triangle 』

You can calculate the area of a triangle if you know the lengths of all three sides, using the **Heron's Formula**, which has been known for nearly 2000 years. Write a program to read **three integers** from the user as the lengths of the three sides of a triangle, and then print the area of the triangle with **2 decimal places**. Use String.format("%.2f", area) to display only 2 decimal places of a double.

Information about Heron's Formula: http://www.mathsisfun.com/geometry/herons-formula.html

The following are examples of input and output:

```
Input:
3 4 5
Output:
6.00

Input:
11 15 19
Output:
82.41
```

Name your program as **AreaOfTriangle**

```java
import java.util.Scanner;

public class AreaOfTriangle {
    public static void main(String[] args) {
        Scanner in;
        in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();
        double s = (a+b+c)/2.0;
        double z = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        System.out.println(String.format("%.2f", z));
    }
}
```

Figure 3.1. User Interface (UI) of Mulberry

Procedures

Overview

This exploratory DRB study consisted of two stages (see Figure 3.2). In the first stage (July 2 to July 15, 2017), students of group 1 took the introductory programming class. Mulberry was integrated into instruction and collected data on student problem-solving attempts. After the first group's course ended, data analysis was conducted to identify common programming misconceptions students demonstrated in their programs to answer RQ 1. In the second stage (July 16 to July 29, 2017), targeted feedback to address these misconceptions was designed using principles from conceptual change and feedback theories (diSessa, 2014; Hattie & Gan, 2011; Vosniadou & Skopeliti, 2014) and added to Mulberry. When students of group 2 took the same introductory programming class and solved programming problems in Mulberry, they received the targeted feedback to address their misconceptions. After the second group's course ended, data analysis was conducted to assess how the feedback affected the evolution of students' (mis)conceptions to answer RQ 2. This study was approved as exempt from Institutional Review Board (IRB).

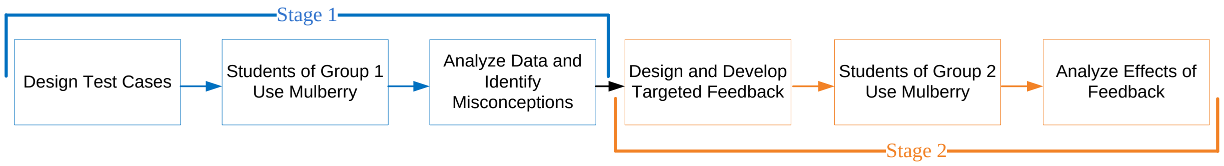

Stage 1
Design Test Cases
Students of Group 1 Use Mulberry
Analyze Data and Identify Misconceptions
Design and Develop Targeted Feedback
Students of Group 2 Use Mulberry
Analyze Effects of Feedback
Stage 2

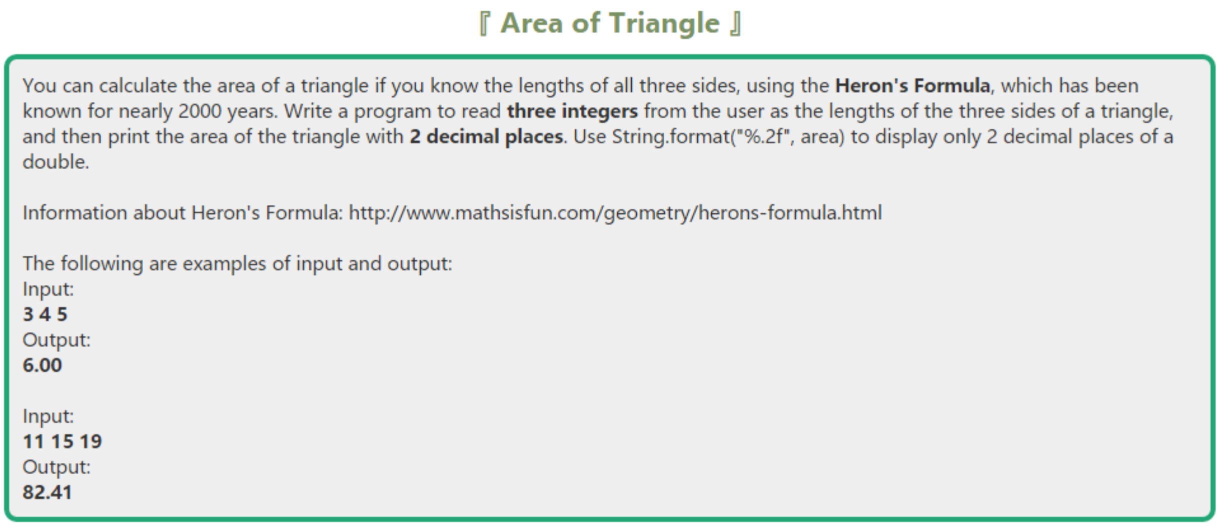
Figure 3.2. Timeline of the study

Stage 1

The goal of the first stage of the study was to identify common programming misconceptions students exhibited in their programs to address RQ 1. In this stage, students of group 1 took the introductory programming class and used Mulberry to practice their programming skills. Each problem in Mulberry was related to one or more of programming concepts covered in the course. Specific test cases of each problem were designed to reveal student misconceptions.

Figure 3.3 shows an example problem in Mulberry and its test cases. In this problem, students needed to write a program that used Heron's Formula to calculate the area of a triangle.

The link to a webpage that explains details about Heron's Formula was given (see Figure 3.4 for a screen shot of the webpage). In addition, the problem description gave examples of test cases which were similar to the real test cases in the backend. When a student submitted a solution to a problem, Mulberry used the test cases of the problem to automatically assess the correctness of the solution by comparing the output of the solution with the expected output. When students had errors in their solutions, they were told that errors existed in their code and were encouraged to try again. Mulberry collected every student's solution no matter whether it was correct or erroneous.



『 Area of Triangle 』

You can calculate the area of a triangle if you know the lengths of all three sides, using the **Heron's Formula**, which has been known for nearly 2000 years. Write a program to read **three integers** from the user as the lengths of the three sides of a triangle, and then print the area of the triangle with **2 decimal places**. Use String.format("%.2f", area) to display only 2 decimal places of a double.

Information about Heron's Formula: http://www.mathsisfun.com/geometry/herons-formula.html

The following are examples of input and output:
Input:
3 4 5
Output:
6.00

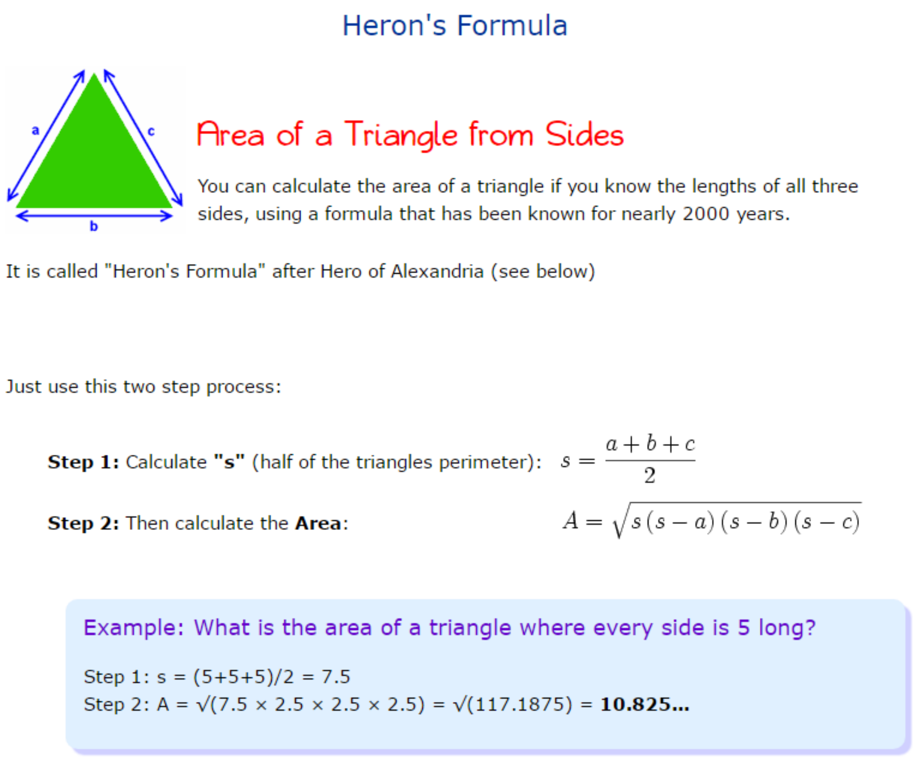
Input:
11 15 19
Output:
82.41

Real Test Cases in the Backend:

Input: 3 4 4	Expected Output: 5.56
Input: 7 8 9	Expected Output: 26.83
Input: 12 13 5	Expected Output: 30.00

Figure 3.3. Example problem **Area of Triangle** and its test cases

When group 1 of summer 2017 completed the course, they produced 695 problem solution attempts in total. Because the number of student solutions in group 1 was relatively small, to obtain a more complete understanding of common student misconceptions, student solutions of group 1 were combined with previous student solutions in Mulberry produced by three groups of students who took this class in summer 2016. The problem solutions generated in summer 2016 came from two groups of Pulsar students and one group was Star students (who had completed grade 7 or 8). In total, these three groups had 42 students who produced 4178 solutions. Thus, for the final analysis of misconceptions, 4873 student solutions from 55 students were pooled for the identification of common student misconceptions. The following section describes details about how common student misconceptions were identified.


Heron's Formula

Area of a Triangle from Sides

You can calculate the area of a triangle if you know the lengths of all three sides, using a formula that has been known for nearly 2000 years.

It is called "Heron's Formula" after Hero of Alexandria (see below)

Just use this two step process:

Step 1: Calculate **"s"** (half of the triangles perimeter): $s = \frac{a+b+c}{2}$

Step 2: Then calculate the **Area**: $A = \sqrt{s(s-a)(s-b)(s-c)}$

Example: What is the area of a triangle where every side is 5 long?

Step 1: s = (5+5+5)/2 = 7.5
Step 2: A = √(7.5 × 2.5 × 2.5 × 2.5) = √(117.1875) = **10.825...**

Link: http://www.mathsisfun.com/geometry/herons-formula.html

Figure 3.4. A screen shot of the webpage explaining Heron's formula

Data Analysis

An erroneous student solution in Mulberry may have had either compilation errors or test errors. When a solution had compilation errors, it failed to compile and produced a compilation error message. In other words, its output was the error message and thus failed to match the expected output. When a student solution was successfully compiled but produced output that did not match the expected output exactly, it was an erroneous solution with test errors. When different students made the same compilation or test error in their solutions, they might have a common misconception. Therefore, analysis was conducted to find the common compilation and test errors first. Next, based on the common errors, common student misconceptions were identified and discussed. As compilation errors are cross-problem while most test errors are problem-specific, common compilation and test errors were analyzed in different ways.

Compilation Errors

A compilation error is a mistake found by the compiler when compiling a program. The Java compiler will produce error messages describing the compilation error(s) a program has. A program may have more than one compilation error. Programs for solving different problems may contain the same compilation errors (e.g., missing semicolons). In this study, when at least 20% of the students showed the same compilation error in their solutions, that compilation error was defined as a common one. In other words, a common compilation error was one that occurred in solutions from at least 11 different students (20% of 55).

After common compilation errors were collected, example student code related to these errors was analyzed to identify common student misconceptions. For instance, one common compilation error is called **possible loss of precision**, which occurs when assigning a value of higher precision to a variable of lower precision. This error indicates students' problematic or incomplete understanding of the concept *Variables*. Students may have the misconception that variables in Java programming are the same as variables in math that do not have a specific type and have unlimited precision. In addition, when several common compilation errors suggested the same misconception, they were combined in the analysis. For example, the common compilation error **reached end of file** is typically caused by a missing closing brace }, and another common compilation error **) expected** is usually caused by a missing closing parenthesis. While they are different errors, they both indicate students' misunderstandings or misapplications of the concept *Program Structure*, specifically *Java Punctuation*.

Test Errors

A test error occurs when a student solution has no compilation error but produces output that does not match the expected output given the designed test cases as the input. Because every problem has its own test cases and expected output, test errors are problem-specific. Figure 3 shows the test cases and expected outputs for each test case of the problem **Area of Triangle**. For example, when the test case (input) is *3 4 4*, a correct student solution should produce the output *5.56*. If the student solution outputs something like *5.562148865321747*, it means that the student solution has a test error that is failing to display only 2 decimal places of the result as described in the problem. When using the other two test cases of the problem **Area of Triangle**, the outputs will be similar (with wrong number of decimal places). These were not considered as different test errors, because they are caused by the same problem of the solution: failing to keep two decimal places of the result. Hence, one erroneous student solution can only have one test error.

For the same problem, sometimes different test errors can be identical in essence. For example, the problem **Say Hi to Anyone** expects students' solutions to produce output like "Hello, Mike!". Some student solutions omitted the comma (,) in the output while a few students omitted the exclamation mark (!). Although the different outputs made them two different test errors, in essence they were identical: missing required punctuation in output. Such test errors were combined in the analysis of common test errors. Moreover, test errors of different problems may also point to the same misconception. For example, failing to keep two decimal places of the result was a common test error in two problems, **Area of Triangle** and **Area of Circle**. In the analysis, these kinds of test errors were treated individually first. They were combined, however, when discussing the underlying misconceptions.

As test errors are related to specific problems, the first step in collecting common test errors was to select *difficult problems*. A *difficult problem* was defined as a problem solved by 50% or more students (at least 28 students) but with a *problem correct rate* lower than 50%. When a student solved a problem, his or her correct rate of solving that problem was named the *student correct rate of problem* and was calculated by the following formula:

$$student\ correct\ rate\ of\ problem\ = \frac{1}{\text{the number of a student's solutions to the problem}}$$

The *problem correct rate* was defined as the correct rate for each problem and was calculated by the following formula:

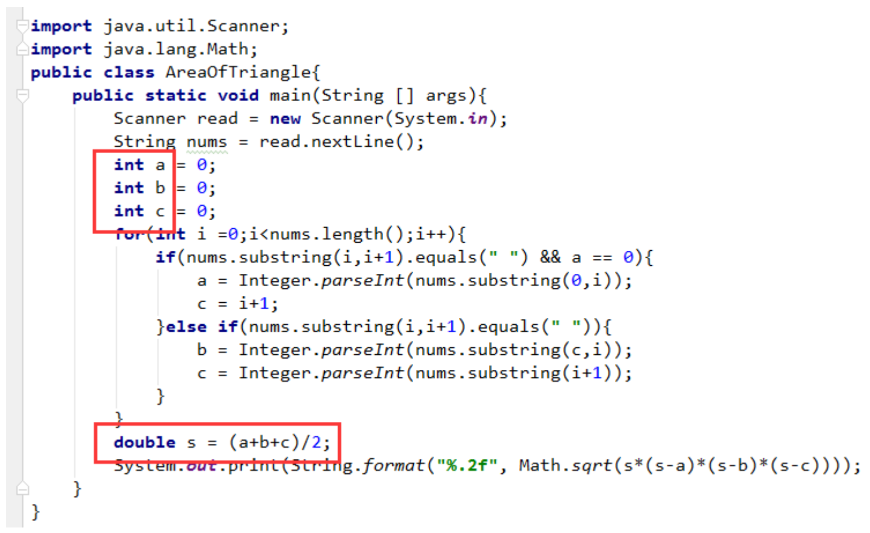
$$problem\ correct\ rate\ = \frac{\sum student\ correct\ rates\ of\ the\ problem}{\text{the number of students who solved the problem}}$$

If a problem was only solved by one or two students, the errors in the students' solutions may not reveal common misconceptions. If a problem was solved by most of the students but with a high *problem correct rate* (e.g., 80%), there may not be enough information for identifying misconceptions. Hence, this exploratory study only focused on *difficult problems*. Using the rules above, the problem **Area of Triangle**, for example, was identified as a *difficult problem*. It was solved by 44 students (80% of the students) with a *problem correct rate* of 34%

(indicating students had many incorrect attempts before solving the problem). In the end, there were 9 *difficult problems* identified in this study (see details in the Results section)**.**

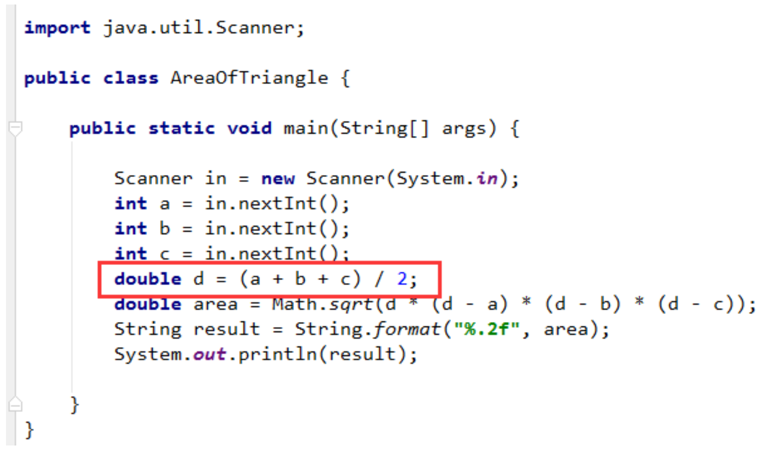
For each difficult problem, a test error was considered common when at least 20% of the students who solved the problem showed the same test error. For example, 44 students solved the **Area of Triangle** problem, and 22 of them (48%) showed the same test error that produced the wrong output *3.16* when the input was *3 4 4*. While students might have written very different incorrect solutions, their key errors were identical. Figure 3.5 illustrates two solutions that were written by two different students but produced the same wrong output *3.16*. In these cases, both students failed to recognize that the result of the expression *(a+b+c) / 2* would be an integer without decimal values. Common test errors appearing in student solutions to difficult problems

Solution #1



```
import java.util.Scanner;
import java.lang.Math;
public class AreaOfTriangle{
    public static void main(String [] args){
        Scanner read = new Scanner(System.in);
        String nums = read.nextLine();
        int a = 0;
        int b = 0;
        int c = 0;
        for(int i =0;i<nums.length();i++){
            if(nums.substring(i,i+1).equals(" ") && a == 0){
                a = Integer.parseInt(nums.substring(0,i));
                c = i+1;
            }else if(nums.substring(i,i+1).equals(" ")){
                b = Integer.parseInt(nums.substring(c,i));
                c = Integer.parseInt(nums.substring(i+1));
            }
        }
        double s = (a+b+c)/2;
        System.out.print(String.format("%.2f", Math.sqrt(s*(s-a)*(s-b)*(s-c))));
    }
}
```

Solution #2



```
import java.util.Scanner;

public class AreaOfTriangle {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();
        double d = (a + b + c) / 2;
        double area = Math.sqrt(d * (d - a) * (d - b) * (d - c));
        String result = String.format("%.2f", area);
        System.out.println(result);

    }
}
```

Figure 3.5. Two different students' solutions producing the same wrong output were used to identify the common misconceptions. For example, the misconception in this example is that students had a problematic understanding of the Java division operator.

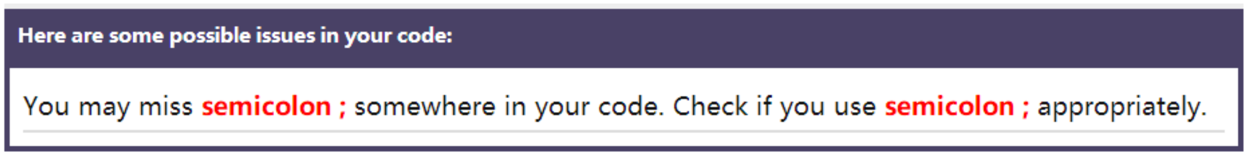
While completed data analysis was not conducted until the first group finished the course, partial data analysis was conducted throughout stage 1. For example, when all the students had solved programming problems about the topic Input/Output (I/O) in Mulberry, partial data analysis of students' solutions to those problems was conducted to estimate potential misconceptions. Potential feedback for addressing these misconceptions was also developed during this process.

Stage 2

The goal of the second stage was to assess how feedback affected the evolution of students' (mis)conceptions to address RQ 2. Before students of the second group started the class, targeted feedback messages to address student misconceptions identified in stage 1 were designed using principles from conceptual change and feedback theories (diSessa, 2014; Hattie & Gan, 2011) and added to Mulberry. Because common student misconceptions were identified based on common compilation and test errors in student solutions, targeted feedback was designed and provided for every common compilation or test error. However, when several common errors were related to the same misconception, the targeted feedback for addressing them was identical or similar. In addition, feedback for addressing common compilation errors and common test errors was also designed differently.

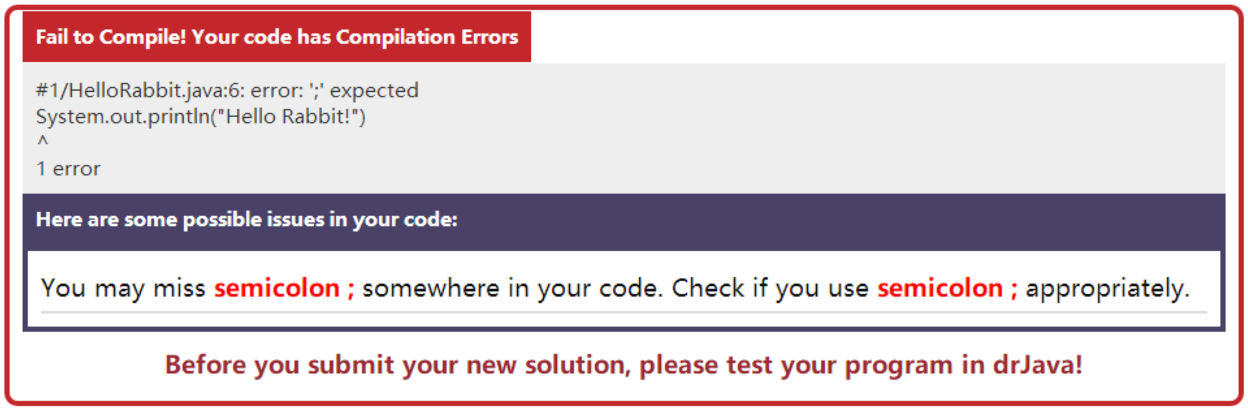
Feedback for Compilation Errors

As compilation errors are not specific to particular problems, targeted feedback for addressing them contained general information about possible problems in the student's solution and potential ways of improving the solution. For instance, the **; expected** error was typically caused by missing required semicolon(s), and students received the following targeted feedback message in Mulberry:


Here are some possible issues in your code:

You may miss **semicolon ;** somewhere in your code. Check if you use **semicolon ;** appropriately.

Because compilation errors can be detected by the IDE students used, drJava, on their local machine, a general feedback message telling students to test their solutions in drJava before submitting them to Mulberry was provided for common compilation errors followed by the targeted feedback message. In addition, the raw error message from the compiler, which was the only feedback message group 1 students received, was also provided. The following is the full feedback message group 2 students received in Mulberry when they had the common compilation error **; expected**.


Fail to Compile! Your code has Compilation Errors

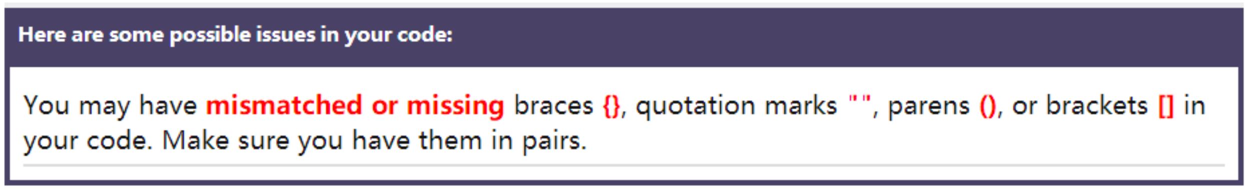
```
#1/HelloRabbit.java:6: error: ';' expected
System.out.println("Hello Rabbit!")
^
1 error
```

Here are some possible issues in your code:

You may miss **semicolon ;** somewhere in your code. Check if you use **semicolon ;** appropriately.

Before you submit your new solution, please test your program in drJava!

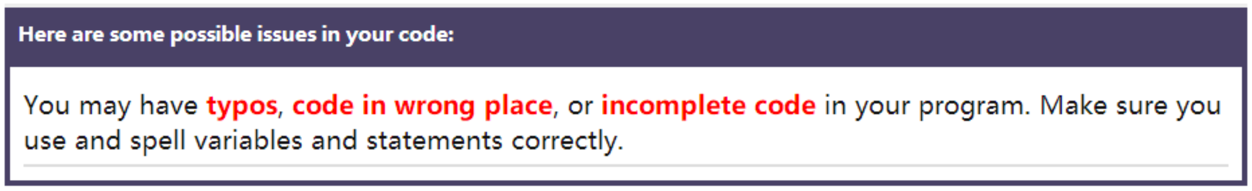
When several common compilation errors were caused by similar mistakes and related to the same misconception, the same feedback message was provided. For example, the common compilation errors **reached end of file** and **) expected** are both about mismatched or missing Java separators that should be used in pairs. Therefore, the following feedback message was provided for both of them as well as two other similar common compilation errors.


Here are some possible issues in your code:

You may have **mismatched or missing** braces **{}**, quotation marks **" "**, parens **()**, or brackets **[]** in your code. Make sure you have them in pairs.

The common compilation error **{ expected** seems to be a similar error, but its cause may be very complicated and often is irrelevant to a missing opening brace {. In order to not mislead students, no targeted feedback was designed and provided for the **{ expected** error.

Finally, compilation errors are not always precisely caught by the compiler and described in the compiler error message. For example, mistakes such as missing a single semicolon, missing braces, or missing the right-hand side of an assignment statement may all result in the common compilation error **illegal start of expression**. More importantly, when this error exists, the compiler error message often points to perfectly good code. Therefore, for such errors, the following general feedback message was provided. In the end, 15 common compilation errors were collected, and eight unique feedback messages were designed for them (See details in the Results section).

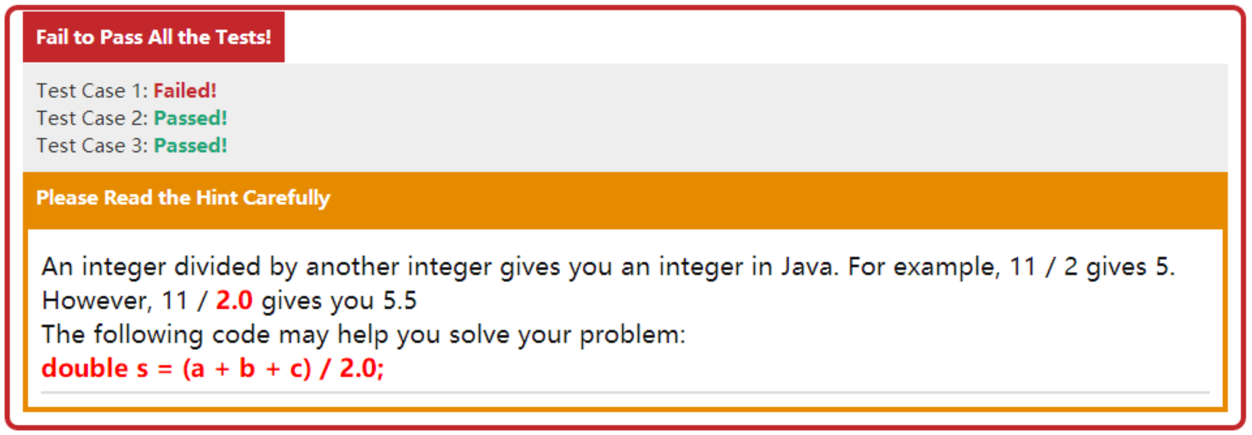


Here are some possible issues in your code:

You may have typos, code in wrong place, or incomplete code in your program. Make sure you use and spell variables and statements correctly.

Feedback for Test Errors

Targeted feedback for addressing common test errors was designed to contain information regarding the specific problem and potential ways of improving the solution. For example, when a student solution to the **Area of Triangle** problem had the common test error illustrated in Figure 3.5, he or she received the following feedback message:



Fail to Pass All the Tests!

Test Case 1: Failed!
Test Case 2: Passed!
Test Case 3: Passed!

Please Read the Hint Carefully

An integer divided by another integer gives you an integer in Java. For example, 11 / 2 gives 5.
However, 11 / 2.0 gives you 5.5
The following code may help you solve your problem:

```
double s = (a + b + c) / 2.0;
```

This feedback message was designed to let students know the current status of their solution and provide guidance about how to fix the error. Other feedback messages for addressing common test errors were designed and provided in a similar way.

Among the 9 *difficult problems*, two of them (the **Arithmetic Operations** problem and the **Sort Three Integers** problem) did not show common test errors, and no feedback was designed for addressing them. The most common test error of the **How Old Are We?** problem, **Mismatched input,** did not meet the common test error standard of this study; however, because its underlying misconception was the same as the common test error **Mismatched input** of the problem **Sum of Digits**, the feedback message for addressing it was added. In the end, 10 common test errors were identified, and 10 unique feedback messages were designed for them (see details in the Results section).

Data Analysis

After targeted feedback was added to Mulberry, students of the second group received feedback messages when their solutions exhibited an identified common error. After the second group's course ended, both quantitative and qualitative data analysis were conducted to see whether and how the targeted feedback made a difference in students' solutions and so may have contributed to conceptual change.

Quantitative Data Analysis

The goal of the quantitative analysis was to check whether the targeted feedback had positive effects on conceptual change. In order to check the effects of feedback, erroneous student solutions of both group 1 and 2 were categorized into two types: *improved* and *not improved*. When the next solution of an erroneous solution for solving the same problem was correct, this means that the student had improved this erroneous solution. Hence, this solution was labelled as *improved*. When an erroneous solution had compilation errors, and its next solution was successfully compiled but failed to pass the test, this also means that the student had improved this erroneous solution, because at least the compilation errors were fixed. Such erroneous solutions were also labelled as *improved.* When an erroneous solution had compilation errors, and its next solution also had compilation errors, it was labelled *not improved*. When an erroneous solution had test errors, and its next solution had compilation or test errors, it was also labelled *not improved*.

After the categorization was done, three different kinds of improvement rates were calculated and compared. First, overall improvement rates of both groups were calculated, which were the proportion of *improved* solutions. Second, each group's improvement rate of solutions with common errors was calculated, which was the proportion of *improved* solutions among the solutions with common errors. Third, for group 2, improvement rates of solutions with and without feedback were calculated, which were the proportion of *improved* solutions among the solutions with and without feedback respectively. Chi-square tests were conducted to see whether the differences in improvement rates were statistically significant.

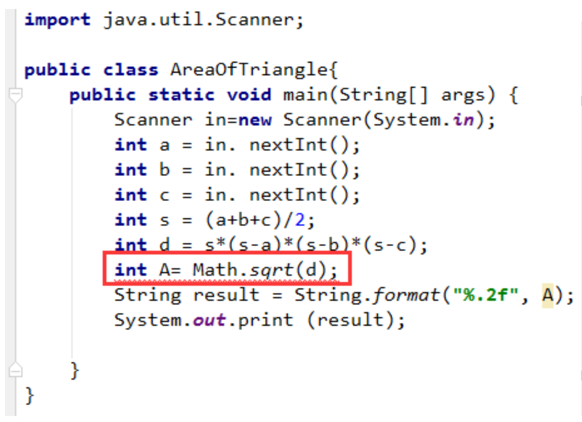
Qualitative Data Analysis

The goal of the qualitative analysis was to understand how targeted feedback affected the evolution of students' (mis)conceptions. Analyzing students' programs qualitatively is vital to complement quantitative analysis and provide further insights into students' conceptual understandings (Fields, Quirke, Amely, & Maughan, 2016). Four feedback cases were selected for the qualitative analysis. The case selection was based on the following procedures. First, for each feedback message, an *improvement rate* was calculated by using the following formula:

$$improvement\ rate = \frac{\text{the number of improved solutions with the feedback}}{\text{the number of occurrences of the feedback}}$$

Feedback messages with the best and worst improvement rate were selected as cases. Cases for compilation errors and test errors were selected separately, so four cases were selected. As certain feedback messages only occurred once or twice, and their improvement rates were either 100% or 0%, case selection only used the feedback messages with an above-average number of occurrences.

When the cases were selected, student solutions of both group 1 and 2 were extracted from the Mulberry database. While students of group 1 did not receive any feedback (other than standard compiler messages), their solutions that had the same error as students in group 2 who got the targeted feedback were used. The patterns of evolution of (mis)conceptions of students from group 1 and group 2 were compared in detail to determine if targeted feedback affected conceptual change as demonstrated via their solutions. Figure 3.6 shows an example of how a student revised his or her solutions to the **Area of Triangle** problem. In Solution #1, the student encountered a syntax error, because the return value of the *Math.sqrt() method* is a double rather than an integer. In Solution #2, the student fixed the syntax error but still mistakenly used integer type variables to store possible double values. In Solution #3, the student figured out the variable type issue but still did not recognize the expression "(a+b+c) / 2" would return an integer and lose precision. Solution #4 is correct. Qualitative analysis like this can lead to an understanding of the evolution of student (mis)conceptions as they actively worked to solve a problem. If targeted feedback were provided when the student submitted Solution #2 and his or her next solution were correct, this suggests that the feedback might have affected conceptual change.

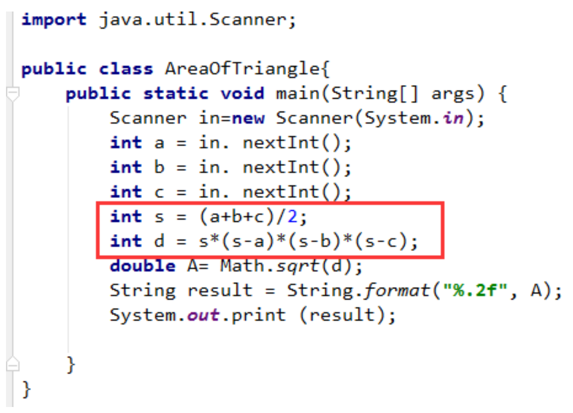


```java
import java.util.Scanner;

public class AreaOfTriangle{
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int a = in. nextInt();
        int b = in. nextInt();
        int c = in. nextInt();
        int s = (a+b+c)/2;
        int d = s*(s-a)*(s-b)*(s-c);
        int A= Math.sqrt(d);
        String result = String.format("%.2f", A);
        System.out.print (result);

    }
}
```

Solution #1

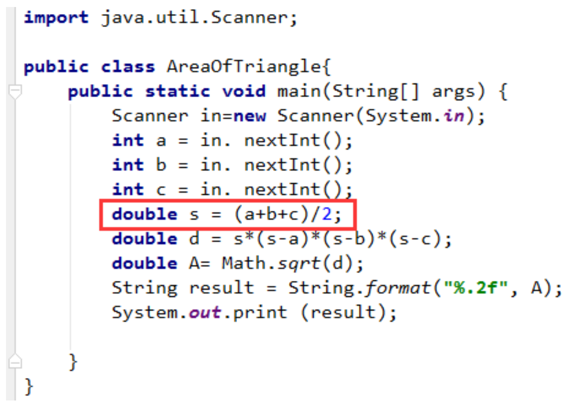


```java
import java.util.Scanner;

public class AreaOfTriangle{
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int a = in. nextInt();
        int b = in. nextInt();
        int c = in. nextInt();
        int s = (a+b+c)/2;
        int d = s*(s-a)*(s-b)*(s-c);
        double A= Math.sqrt(d);
        String result = String.format("%.2f", A);
        System.out.print (result);

    }
}
```

Solution #2

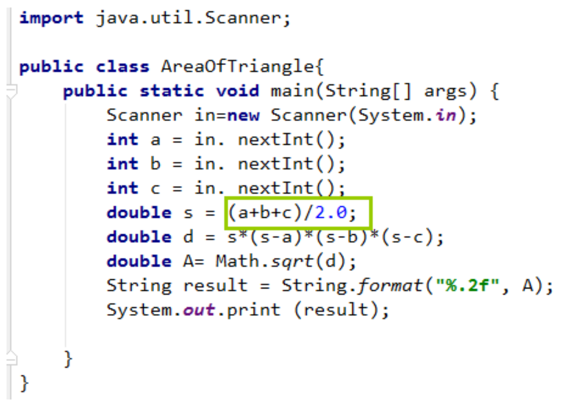


```java
import java.util.Scanner;

public class AreaOfTriangle{
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int a = in. nextInt();
        int b = in. nextInt();
        int c = in. nextInt();
        double s = (a+b+c)/2;
        double d = s*(s-a)*(s-b)*(s-c);
        double A= Math.sqrt(d);
        String result = String.format("%.2f", A);
        System.out.print (result);

    }
}
```

Solution #3



```java
import java.util.Scanner;

public class AreaOfTriangle{
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        int a = in. nextInt();
        int b = in. nextInt();
        int c = in. nextInt();
        double s = (a+b+c)/2.0;
        double d = s*(s-a)*(s-b)*(s-c);
        double A= Math.sqrt(d);
        String result = String.format("%.2f", A);
        System.out.print (result);

    }
}
```

Solution #4

Figure 3.6. A student's solutions to the **Area of Triangle** problem

CHAPTER 4: RESULTS

Identification of Misconceptions

Common Compilation Errors

To address RQ 1, students' erroneous problem solutions were analyzed to identify common errors and specific misconceptions. Among the 4873 student solutions from the 55 students (13 students of group 1 and 42 students of summer 2016), 3632 solutions were incorrect. Solutions with compilation errors numbered 1752, and solutions with test errors were 1880. Among the 1752 compilation-erroneous solutions, there existed 2335 compilation errors. By grouping the same compilation errors, 55 distinct compilation errors were identified, and 15 of them were categorized as common ones. The 15 common compilation errors occurred 2151 times in total and accounted for 92% of all compilation errors. Table 4.1 presents these common compilation errors. The "CE" in the error number stands for "Compilation Error." Additional information in the table includes the error name, and the occurrence rate. The occurrence rate is the number of students who made the error and the percentage out of the 55 total students.

Four error names were simplified from the actual compiler error messages. The **program name error** was originally described as something like "public class Abc should be in a file named Abc.java." It was caused by the mismatch between the program's name (the name of the class) and the program file name, so it was renamed into the simpler version "program name error." The **class expected** error was originally called "class, interface, or enum expected." As interfaces and enums were not introduced and used in this course, it was renamed into "class expected" for short. The **reached end of file** error was a short version of the original error message "reached end of file while parsing." The **incorrect use of operators** error was a revision of the original compiler error message "bad operand types for binary operator."

Based on the 15 errors, three common misconceptions were identified. The following section provides the details about the common compilation errors and underlying misconceptions.

Table 4.1

Common Compilation Errors

#	Error	Occurrence
CE1	cannot find symbol	50/55 (91%)
CE2	; expected	48/55 (87%)
CE3	program name error	48/55 (87%)
CE4	class expected	40/55 (73%)
CE5	reached end of file	37/55 (67%)
CE6	not a statement	31/55 (56%)
CE7	) expected	29/55 (52%)
CE8	illegal start of expression	29/55 (52%)
CE9	identifier expected	24/55 (44%)
CE10	incompatible types	23/55 (42%)
CE11	variable is already defined	19/55 (35%)
CE12	incorrect use of operators	15/55 (27%)
CE13	illegal start of type	13/55 (24%)
CE14	{ expected	13/55 (24%)
CE15	possible loss of precision	13/55 (24%)

```java
import java.util.Scanner;

public class ExampleOne {
    public static void main(String[] args) {

        Scanner sc;
        sc = new Scanner(System.in);

        int a = sc.nextInt();

        if(a > 0) {
            System.out.println(a + " is a positive number.");
        }
        else {
            System.out.println(a + " is not a positive number.");
        }
    }
}
```

Figure 4.1. An example Java program

Misconception 1: Deficient Knowledge of Fundamental Java Program Structure

Seven of the 15 common compilation errors were related to fundamental Java program structure, including CE2, CE3, CE4, CE5, CE7, CE9, and CE14. To explain these errors clearly, Figure 4.1 shows an example Java program, which reads an integer as input and decides if the integer is positive.

CE2: **; expected** typically resulted from the missing semicolon (;) at the end of a statement. In Java, every statement must end with a semicolon (see line 9 in Figure 4.1 for an example). However, many students forgot this convention when writing programs and failed to include the necessary semicolons.

CE3: **program name error** occurred when a student program's name (the name of the class) did not match its file name. A Java program must define a *class* and be saved in a file with the same name as the *class* name followed by the extension *.java*. The example program in Figure 4.1 defines a *class* called "ExampleOne", so it should be saved in a file called "ExampleOne.java". When solving a problem in Mulberry, the required program name was given in the problem description. However, when submitting solutions, many students did not use the required class name or misspelled the class name. For example, the required program name of the problem **Area of Triangle** was "AreaOfTriangle", but several students used "AreaofTriangle". Though these two names look similar, Java is case sensitive and requires an exact match of the program name and the file name.

Three errors (CE4, CE5, and CE14) were related to unbalanced braces {}. In Java programming, braces are used to group statements into code blocks. For example, the opening and closing braces in line 3 and 18 in Figure 4.1 enclose all the statements of the *class ExampleOne* and define the *class* content block. CE4: **class expected** indicated an incorrect *class* definition and was mainly caused by a missing opening brace { in the *class* definition line. As the Java compiler was expecting a correct *class* definition but an incorrect one was given, it reported this error. On the contrary, CE5: **reached end of file** typically resulted from a missing closing brace }. As the compiler could not find the closing brace the end of the program, this error occurred. CE14: **{ expected** indicated a missing opening brace in the code. In this study, only a few times was this error truly caused by a missing opening brace. Many times, it resulted from having redundant punctuation when defining a *class*. Some students had punctuation marks such as comma and space in the class name. A few students added punctuation marks such as a semicolon after the class name. In all these cases, the Java compiler interpreted the code as meaning that the class name ended before those punctuation marks and was expecting an opening brace {, and thus CE14 occurred.

CE7: **) expected** was mainly caused by missing the closing parenthesis). In Java, parentheses () are used to control the order of expression evaluation or enclose parameters of a method. Parentheses marks must be used in pairs. However, students in this study often omitted the closing parenthesis in their code. This error sometimes also occurred when the parentheses were balanced but the expression within the parentheses was erroneous. For example, line 2 and 3 in Figure 4.2 also led to CE7: **) expected**, because the Java compiler could not evaluate the expression after the opening parenthesis and failed to locate the closing parenthesis.

CE9: **identifier expected** indicated that an identifier did not appear where it should. In Java, identifiers are the names used as labels, including variable names, class names, method names, and so forth. While this seemed to be relevant to missing identifiers, in this study, most of the time it was caused by unbalanced braces. A Java program may have some data members (e.g., a class variable) and methods (e.g., the main method). Identifiers are necessary to define the data members and methods. In Figure 4.1, line 4 defines the *main method* of the *class*. The opening brace in the method definition line indicates the start of the main method, and the closing brace in line 17 means the end of the method block. When a student missed the opening brace of the main method, the compiler interpreted all the code within the main method block as statements for defining data members or methods. Thus, the compiler was expecting identifiers, but other types of code were given (e.g., a method call *System.out.println*). Omitting the opening brace of a code block (e.g., the opening brace in line 11 of Figure 4.1) or having extra closing braces somewhere in the main method might also lead to this error. In both cases, an extra closing brace mistakenly indicated the end of the main method block and made code after it part of other data member and method definitions. Hence, CE9 occurred when the complier could not find the expected identifiers. In addition, several students completely omitted the *main* method in their program and directly enclosed statements in the *class* definition block. This also produced this error.

Misconception 2: Misunderstandings of Java Expressions

Three of the 15 common compilation errors were related to incorrect use of Java expressions, including CE6, CE8, and CE13. A Java expression is made up of variables, operators, and method calls, and evaluates to a single value. Expressions are major components of statements. In this study, students often constructed expressions without following Java proper syntax.

```
1   //Some lines have more than one error
2   System.out.println(Don't Worry!);
3   System.out.println(m "+" n "=" + sum);
4
5   String final = String.format("%.2f", area);
6   double 3.14 = pi;
7   double = (n+m);
8
9   int a = [(n-n/3)*3]/n;
10  d = a + 0.1a + 0.01a;
11
12  String result = String.format("%.2f, area);
13  System.out.println(result);
```

Figure 4.2. Examples of erroneous code about expressions

CE6: **not a statement** error occurred when the compiler was expecting a syntactically correct statement but something else was given. It typically resulted from syntactically incorrect expressions within a statement. Figure 4.2 presents several examples of this error from students' code. In line 2, the expression within the parentheses is wrong because this student forgot to put the output message "Don't Worry!" within quotation marks. In line 3, the expression within the parentheses could not be evaluated to a value as it failed to concatenate the variables m and n and the string literal *"+"* using plus signs (+). In line 5, the Java keyword *final* was used as the variable name in the assignment expression which is not allowed in Java. In line 6, the student reversed the order of the value (3.14) and the variable (pi) in the assignment expression. In line 7, the variable name of the assignment statement was missing which made the statement incomplete. Code in line 9 and 10 made the same error: using illegal mathematical notations in Java expressions. While brackets may be used to enclose parentheses in mathematical expressions, brackets and parentheses in Java have different meanings, and extra layers of parentheses should be used when necessary. In line 9, brackets should be replaced by parentheses to make the expression correct. In line 10, the multiplication operator (*) must be included even though it would not be required in a math expression.

CE8: **illegal start of expression** usually resulted from having erroneous expressions in the code. Line 5 and 9 in Figure 4.2 are two examples of illegal expressions. This error also occurred when a correct expression was placed in a wrong location. For example, the *import* statement should be put before the class definition (see line 1 in Figure 4.1), but a few students placed it in the class block. While they wrote the *import* statement correctly, the Java compiler considered it as an inappropriate statement. Another situation that led to this error was that something was wrong (e.g., unclosed quotation marks, parentheses, and braces) before a correct expression. In line 12 of Figure 4.2, this student forgot the closing quotation mark, and thus the closing parenthesis and the semicolon were considered as part of the string literal by the compiler. While the next line (line 13) had a correct expression, the error occurred because the compiler was expecting something matching the previous line.

CE13: **illegal start of type** usually was a side effect of the previous two errors. For example, when a Java keyword was used as the variable name in assignment (see line 5 in Figure 4.2) or the *import* statement was placed within the class definition, this error also occurred. Sometimes, it was caused by mismatched braces or parentheses. For instance, if the opening brace in line 11 of Figure 4.1 is omitted, the compiler will consider the keyword *else* in line 14 as the start of type, which is illegal.

Misconception 3: Confusion about Java Variables

Five of the 15 common compilation errors were related to deficient knowledge or misunderstandings of Java variables and variable operations, including CE1, CE10, CE11, CE12, and CE15.

CE1: **cannot find symbol** was the most common error in this study. This error occurred when an identifier, typically a variable, was not declared before being used in the program. Declaration of variables before using them is required in Java. However, in this study, students often forgot variable declaration. This result is not surprising because students do not have to declare variables in math, which is where most students first learned the concept *Variable*. Another cause of this error was the incorrect spelling of variable names which led to inconsistencies between identifiers' declaration and use. Java identifiers are case sensitive, but many students used lower-case and upper-case letters interchangeably (see line 2 and 3 in Figure 4.3). Some students defined a variable using one name (e.g., n) but used another name to refer to the variable (e.g., a).

```
1   //cannot find symbol
2   int sum = 0;
3   Sum = Sum + 1;
4   
5   //variable is already defined
6   int result = 0;
7   String result = String.format("%.2f", area);
8   
9   // incompatible types
10  int n = sc.nextLine();
11  String name = sc.nextInt();
12  
13  int a = 10;
14  int b = 20;
15  if(a = b){...}
16  
17  //incorrect use of operators
18  String i = in.nextLine();
19  String j = in.nextLine();
20  String k = i%j;
21  
22  double ans = -b+((b^2)-4*a*c)^.5;
23  
24  if(a>=b>=c){...} //variables a, b, and c are of int type
25  
26  //possible loss of precision
27  int radius = in.nextDouble();
28  int s = (a+b+c)/2.0;
29  int s = Math.sqrt(n);
```

Figure 4.3. Examples of erroneous code about variables and variable operations

On the contrary, CE11: **variable is already defined** was due to duplicate variable names. In Java, every variable in the same scope needs to have a distinct name (identifier). When two variables were declared using the same name, this error occurred. Line 6 and 7 in Figure 4.3 presents an example of this error.

CE10: **incompatible types** occurred when assigning a value of an expression to a variable whose type was incompatible with the expression type. The analysis of students' code showed that this error usually appeared in two situations. First, students' programs often read input in one type and assigned the input value to a variable with an incompatible type. For example, in line 10 of Figure 4.3, the *sc.nextLine()* read *String* input but its value was assigned to an *int* variable. Line 11 of Figure 4.3 did it in a reversed way: reading *int* value and assigning it to a *String* variable. The second situation was related to the if-statement (conditional statement). Many students used the assignment operator (=) instead of the equality operator (==) to compare the values in the condition expression (see the expression in the parentheses in line 15). In Java, the condition expression in an if-statement must be a boolean expression, which evaluates to type boolean. However, because of the use of the assignment operator, the expression in this example resulted in an *int* value and thus CE10 occurred.

CE12: **incorrect use of operators** resulted from using an operator for operands whose types were not allowed for this operator. One common example was that students used *String* type variables as operands for the modulus operator % (see line 18 to 20 in Figure 4.3). In Java, the modulus operator takes *int* type operands and returns the remainder. However, some students did not notice the variable type issues when using the modulus operator. Another common cause of this error was using Java operators in a mathematical way (see line 22 and 24 for examples). In line 22, the student wanted to use the ^ operator to get the power of expressions, in particular, *b* squared and the square root (the power of one-half) of the expression in parentheses. While the ^ operator may work as the exponentiation operator in math, it is the XOR operator in Java, and thus the operator was incorrectly used here. In line 24, comparisons were chained, which is allowed in math. However, Java evaluates comparisons one by one. In this case, the expression *a>=b* evaluated to a boolean value, which was then compared to the variable *c* using the >= (greater than or equal to) operator. Hence, the comparison operator was not used correctly, because comparing a boolean with an *int* was not allowed by the >= (greater than or equal to) operator.

CE15: **possible loss of precision** typically occurred when students tried to assign a *double* type value (higher precision) to an *int* type variable (lower precision). Figure 4.3 presents two examples of this error. In line 27, this student tried to read the input value of the *double* type using the *Scanner* and assigned to an *int* type variable *radius*. In line 28, the expression on the right side of the equals sign evaluated to a double type value but the variable on the left side of the equals sign was of *int* type.

Common Test Errors

According to the selection standards of difficult problems, nine problems were identified as difficult problems (see Table 4.2). Two difficult problems, **Sort Three Integers** and **Arithmetic Operations**, did not have any errors that met the common test error selection standards. Among the other seven difficult problems, 10 common test errors were found. Among all the test errors, 54% were related to those difficult problems, and the 10 common test errors accounted for 39% of all test errors of the difficult problems. The 10 common test errors are numbered and presented in Table 4.3. The "TE" in the error number stands for "Test Error". Additional information in the table includes relevant problems, the error name, and the occurrence rate. Four common misconceptions were identified based the 10 test errors. Details of the errors and the misconceptions are provided and discussed in the following section.

Table 4.2

Difficult Problems

Problem	Correct Rate	Solved
Area of Circle	27%	52/55 (95%)
Say Hi to Anyone	28%	55/55 (100%)
Area of Triangle	34%	44/55 (80%)
Quadratic Equation 2	41%	34/55 (62%)
Sort Three Integers	41%	37/55 (67%)
Sum of Digits	42%	50/55 (91%)
Arithmetic Operations	47%	49/55 (89%)
How Old Are We?	47%	55/55 (100%)
Who is Max?	48%	42/55 (76%)

Table 4.3

Common Test Errors

Problem	#	Test Error	Occurrence Rate
Area of Circle	TE1	Mismatched input	28/52 (54%)
	TE2	Wrong decimal places	15/52 (29%)
Say Hi to Anyone	TE3	Missing punctuation	23/55 (42%)
Area of Triangle	TE4	Integer division issue	22/44 (50%)
	TE5	Wrong decimal places	11/44 (25%)
Quadratic Equation 2	TE6	Inappropriate comparison	12/34 (35%)
	TE7	Wrong output	7/34(21%)
Sum of Digits	TE8	Mismatched input	16/50 (32%)
How Old Are We?	TE9	Mismatched input	6/55 (11%)
Who is Max?	TE10	Forgot Special Cases	18/42 (43%)

Misconception 1: Misunderstandings of Java Input

Among the 10 common test errors, three were related to mismatched input (TE1, TE8, and TE9). In the problem **Area of Circle**, the student program had to read the radius and calculate the area of the circle. The given radius was a number, either an integer or a decimal number. However, twenty-eight students used the *nextInt()* method of the Scanner to read the input. As this method could only read integer input, when the given radius was a decimal number (e.g., 5.9) the mismatched input error occurred. In the **Sum of Digits** problem, students were required to write a program to read a 3-digit integer and calculate the sum of the digits using the modulus operator and the division operator. Sixteen students made a mistake that called the *nextInt()* method three times to read three integers instead of one integer. As only one integer was given as input, calling the *nextInt()* method three times made the input mismatched. In the **How Old Are We?** problem, the student program had to read two integers as input and calculate the sum. Some students called the *nextInt()* method too many times in their solutions or called the *nextLine()* method twice to read two Strings rather than integers.

Misconception 2: Misunderstandings of Java Output

Four common test errors were related to incorrect output (TE2, TE 3, TE 5, and TE7). Among the four errors, two of them (TE2 and TE5) were the same: forgetting to keep 2 decimal places of the output as required by the problem. In the problems **Area of Circle** and **Area of Triangle**, students were required to output the area as *double* type with only two decimal places. The statement for keeping 2 decimal places *String.format("%.2f", area)* was given in the problem description, and similar examples were introduced in class. Still many students forgot to keep 2 decimal places of the output or did not store the result in the variable that was output. In the problem **Say Hi to Anyone**, the student program had to read a name such as “Mike” and output a sentence like “Hello, Mike!”. Between the word “Hello” and the input name, there is a comma followed by a space. At the end of the sentence, there is an exclamation mark (!). Many student solutions omitted the required punctuation marks in the output. Possibly they mistakenly believed that the output of the solution did not have to exactly match the expected output. Another output-related error occurred when students solved the problem **Quadratic Equation 2**. Students were required to write a program to solve standard-form quadratic equations. When the equation has two roots, the correct program should output the smaller one on the first line and the larger one on the second line. However, some students did not output the roots in the required order or output the roots in the same line.

Misconception 3: Confusion about Java Operators

Two common test errors were related to inappropriate use of operators. For the problem **Area of Triangle**, students were required to use Heron's formula to calculate the area of a triangle given three integers (say a, b, and c) which represented the lengths of the three sides. The first step of applying Heron's formula is to calculate s (half of the triangle’s perimeter), by using the expression: *s = (a+b+c)/2*. In Java, when the two operands of the division operator are integers (*int* type), integer division is used and returns an *int* type value. For instance, the result of 11/2 is 5 rather than 5.5. On the other hand, when either operand is a double, the result of the division will be a double type value. For example, the result of 11/2.0 is 5.5. While this feature of the division operator was illustrated and discussed during the class, many students did not notice that when the sum of a, b, and c was an odd number the result of the expression *(a+b+c)/2* would be an integer and ignored the decimal places. Thus, the final result of the solution -- the area of the triangle -- was incorrect. The other error of inappropriate operator use occurred when solving the problem **Quadratic Equation 2**. Many students used the Java equality operator == to compare doubles or Strings. As double type values are not exact in Java, using the equality operator to compare two doubles might lead to a wrong result. Furthermore, because the problem required displaying two decimal places of the output (one root or two roots), many students formatted the roots into Strings and used the equality operator to check if the two roots were equal. These students believed that when the two roots were equal the equation had one root. However, in Java, the equality operator does not work for comparing Strings; the correct way is to call the *equals()* method to check equality of two String values. In this case, even though the logic of students’ solutions seemed to be correct for solving the problem, the output failed to match the expected one given certain test cases.

Misconception 4: Forgetting to Consider Special Cases

The last common test error occurred when solving the problem **Who is Max?**. To solve this problem, students were required to write a program to find the greatest number among three given integers. The most common issue here was that many students failed to consider special cases of the given integers. When two or three integers were equal, many student solutions did not produce any output, because their conditional statements did not consider such cases. As novices, it is not surprising that they forgot to inspect boundary conditions and/or special cases.

Overall Effects of Feedback

To address RQ2, both quantitative and qualitative data analysis were conducted to see whether and how the targeted feedback made a difference in students' solutions and so may have contributed to conceptual change. This section presents the results of the quantitative analysis which checked whether the targeted feedback had positive effects on conceptual change.

Difference in Overall Improvement Rates

Students' solutions of the two groups were used to analyze the effects of feedback. In total, group 1 and group 2 made 529 and 399 erroneous solutions respectively. When calculating the improvement rate, student solutions with no "next solution" were excluded from the analysis, because without a "next solution" the improvement of an erroneous solution could not be determined. In the end, group 1 had 521 erroneous solutions, and 176 of them were improved. Group 2 had 397 erroneous solutions, and 177 of them were improved. Thus, the improvement rates of the two groups were 34% and 45% respectively (see Figure 4.4). A chi-square test was performed to examine the relationship between group and improvement rate. The improvement rates of the two groups were significantly different, χ^2 (1, N = 918) = 11.11, $p < .001$. Overall, students of group 2 were more likely to improve their erroneous solutions than those of group 1.

Difference in Improvement Rates of Solutions with Common Errors

As students of group 2 received targeted feedback when their solutions had common errors, it was expected that students of group 2 would have a better improvement rate of solutions with common errors than students of group 1. Among the 521 erroneous solutions of group 1, 310 had common errors, and 119 of them were improved. Among the 397 erroneous solutions of group 2, 170 solutions showed common errors and received feedback, and 99 of them were improved. Hence, the two groups' improvement rates of solutions with common errors were 38% and 58% respectively (see Figure 4.5). The results of a chi-square test indicated that the difference in improvement rates was significant, χ^2 (1, N = 480) = 17.45, p < .001. The results suggest that when a student solution had common errors, a student who received targeted feedback was more likely to effectively improve his or her solution.

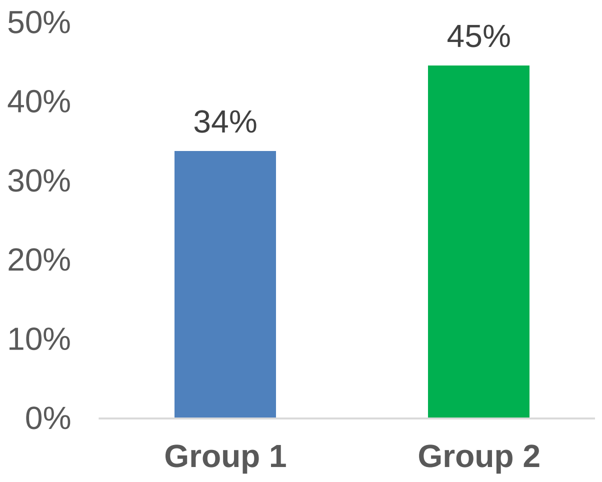


Figure 4.4. Overall improvement rates

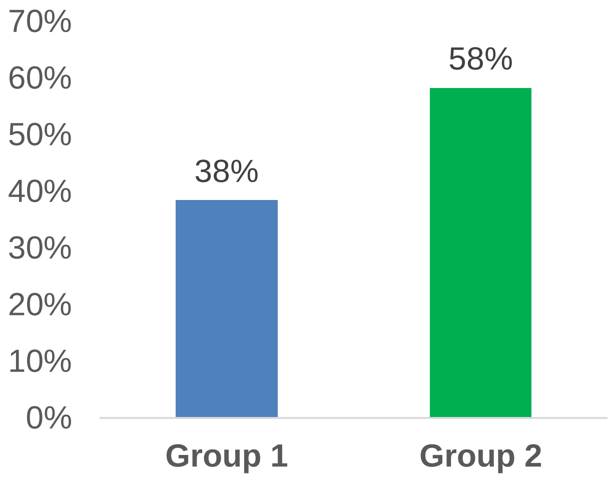


Figure 4.5. Improvement rates of common errors

Difference in Improvement Rates of Solutions with and without Feedback

For group 2, while 170 erroneous student solutions received feedback, the other 227 solutions, which had non-common errors, did not receive targeted feedback. It was also expected that solutions with feedback would have a better improvement rate than those with no feedback. The results confirmed the hypothesis. For solutions with targeted feedback, 99 were improved with an improvement rate of 58%; for solutions without targeted feedback, 78 were improved with an improvement rate of 34% (see Figure 4.6). The results of a chi-square test indicated that the difference was significant, χ^2 (1, N = 397) = 22.42, p < .001. In other words, when a feedback message was presented, a student of group 2 was more likely to effectively improve his or her erroneous solution.

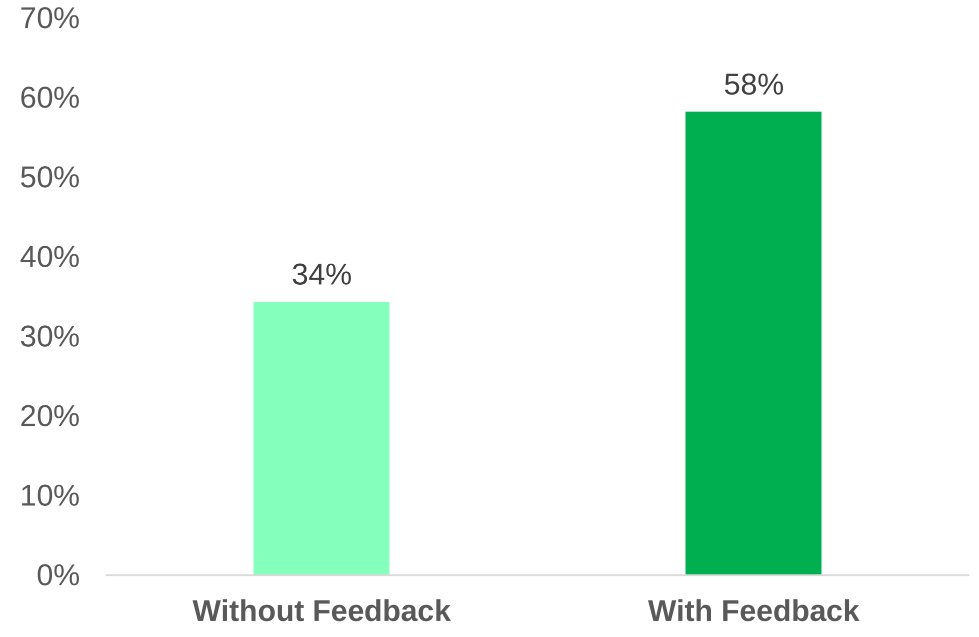


Figure 4.6. Improvement rates of group 2's solutions without and with feedback

Effects of Feedback on Evolution of Students' Misconceptions

Qualitative analysis of student code was also used to address RQ2. The goal of the qualitative analysis was to understand how targeted feedback affected the evolution of students' (mis)conceptions. Four feedback cases were selected for the qualitative analysis according to the case selection procedures described in the Methodology section. First, improvement rates of each feedback message were calculated. Table 4.4 and Table 4.5 present the improvement rates of feedback for compilation and test errors respectively. In addition to the improvement rate, the number of occurrences of the errors and the number of improvements are also included in the tables (see the numbers within the parentheses). While students of group 1 did not receive targeted feedback messages for the common errors, their improvement rates are also presented in the tables in order to make comparisons. As no student in group 2 made the common test errors TE6 and TE9, relevant feedback messages TFB6 and TFB9 are not included in Table 4.5. See Appendix B for detailed information of each feedback message.

Table 4.4

Feedback for Compilation Errors

Feedback	Relevant Errors	Improvement Rate	
		Group 2	Group 1
CFB1	CE4: class expected CE5: reached end of file CE7:) expected CE9: identifier expected	50% (18/36)	34% (26/76)
CFB2	CE1: cannot find symbol CE6: not a statement CE8: illegal start of expression CE13: illegal start of type	61% (35/57)	45% (35/77)
CFB3	CE2: ; expected	43% (10/23) *	45% (18/40)
CFB4	CE3: program name error	83% (25/30) **	54% (30/56)
CFB5	CE10: incompatible types	100% (1/1)	100% (2/2)
CFB6	CE11: variable is already defined	63% (5/8)	50% (3/6)
CFB7	CE12: incorrect use of operators	67% (2/3)	50% (1/2)
CFB8	CE15: possible loss of precision	50% (1/2)	75% (3/4)

Note. * indicates the feedback with the worst improvement rate, and ** indicates the feedback with the best improvement rate. Average number of occurrences was 20.

Table 4.5

Feedback for Test Errors

Feedback	Relevant Errors	Improvement Rate	
		Group 2	Group 1
TFB1	TE1: Mismatched input	17% (1/6) *	11% (3/28)
TFB2	TE2: Wrong decimal places	50% (3/6)	67% (2/3)
TFB3	TE3: Missing punctuation	50% (3/6)	50% (6/12)
TFB4	TE4: Integer division issue	83% (5/6) **	50% (2/4)
TFB5	TE5: Wrong decimal places	33% (1/3)	29% (2/7)
TFB7	TE7: Wrong output	100% (1/1)	100% (1/1)
TFB8	TE8: Mismatched input	25% (1/4)	33% (1/3)
TFB10	TE10: Forgot Special Cases	20% (1/5)	14% (2/14)

Note. * indicates the feedback with the worst improvement rate, and ** indicates the feedback with the best improvement rate. Average number of occurrences was 4.63.

According to the case selection procedures, the average number of occurrences of common compilation and test errors were calculated, which were 20 and 4.63 respectively. Thus, the four selected cases were CFB4, CFB3, TFB4, and TFB1 because the goal was to identify extreme cases (best and worst rates of improvement) with an above average number of occurrences. CFB4 and CFB3 were the compilation error feedback messages (CE Feedback) with the best and worst improvement rate (IR). TFB4 and TFB1 were the test error feedback messages (TE Feedback) with the best and worst improvement rate (IR). Table 4.6 presents the details about the four feedback messages.

Table 4.6

Selected Feedback Cases

Type	IR	Content
CE Feedback with **Best** IR	83%	The name of your program is wrong! Please name your program as XXX!
CE Feedback with **Worst** IR	43%	You may miss **semicolon ;** somewhere in your code. Check if you use **semicolon ;** appropriately.
TE Feedback with **Best** IR	83%	An integer divided by another integer gives you an integer in Java. For example, 11 / 2 gives 5. However, 11 / **2.0** gives you 5.5 The following code may help you solve your problem: `double s = (a + b + c) / 2.0;`

TE Feedback with **Worst** IR	17%	The user may enter a number such as 2.3. Your program has to read a **double** instead of an **int**.
		The following code may help you solve your problem:
		`Scanner in = new Scanner(System.in);`
		`double radius = in.nextDouble();`

Note. XXX will be replaced by the required program name of a problem.

Compilation Error Feedback Message with Best Improvement Rate

The feedback message for addressing the **program name error** showed an improvement rate of 83%. The **program name error** was a straightforward compilation error which occurred when a student program's name (the name of the class) did not match its file name. In Mulberry, the required program name was provided in the problem description. However, students often forgot to use the required name or misspelled the program name. The targeted feedback message for addressing this error was also straightforward and described what was wrong and provided the correct program name (see Table 4.6). The analysis of relevant student solutions found different patterns of improving the code between students of group 2 and group 1.

#	Solution #1	Solution #2
1	`//SumOfTwo//`	`//SumOfTwo//`
2		
3	`import java.util.Scanner;`	`import java.util.Scanner;`
4		
5	`public class Problemsolving {`	`public class SumOfTwo {`
6	`  public static void main (String[]args){`	`  public static void main (String[]args){`
7		
8	`    Student code were hidden`	`    Student code were hidden`
9	`  }`	`  }`
10	`}`	`}`

Figure 4.7a. Group 2 student code example of improvement of program name error

#	Solution #1	Solution #2
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2		
3	`public class Project01 {`	~~`public class Project01 {`~~
4	`  public static void main (String[] args) {`	`  public static void main (String[] args) {`
5		
6	`    Student code were hidden`	`    Student code were hidden`
7	`  }`	`  }`
8	`}`	`}`

#	Solution #3	
1	`import java.util.Scanner;`	
2		
3	`public class HelloAnyone {`	
4	`  public static void main (String[] args) {`	
5		
6	`    Student code were hidden`	
7	`  }`	
8	`}`	

Figure 4.7b. Group 1 student code example of improvement of program name error

When students of group 2 had the **program name error** in their solutions, they typically could directly locate the error and revise the program name into the correct one. Figure 4.7a presents a typical code improvement scenario of students in group 2. This student first named the class as *Problemsolving* and submitted the solution. In the next solution, this student revised the class name into the correct one *SumOfTwo*. On the contrary, students in group 1 often had intermediate solutions to fix this error. Figure 4.7b presents a student case. In the first solution, this student named the class as *Project01* while the required program name was *HelloAnyone*.

Next, rather than revising the class name, this student deleted the whole class definition line. Because students of group 1 only received the default compiler message as feedback, this student might see an error message like “Error: class Project01 is public, should be declared in a file named Project01.java.” With this error message, as a novice, students might not be able to understand what exactly was wrong. In this case, this student might mistakenly believe that the line *public class Project01* was wrong and should be deleted. Finally, in the third solution, he or she realized that it was a **program name error** and fixed the problem.

While this feedback message was simple, it helped students understand what was wrong with the program and how to fix it. The program naming rule was introduced in the class and repeatedly practiced during problem solving. Hence, students probably knew this rule. However, without a targeted feedback message, they might have difficulties to understand or notice the error. The feedback helped to reduce the number of intermediate solutions during the code improvement process.

Compilation Error Feedback Message with Worst Improvement Rate

The feedback message for addressing the **; expected** error was identified as the worst case. The improvement rate of group 2 (43%) was even less than that of group 1 (45%). This feedback message seemed to be relatively ineffective. However, the qualitative analysis of student code revealed that the quantitative analysis failed to identify all the improved cases.

#	Solution #1	Solution #2
1	`//Saymore`	`//Saymore`
2	`import java.util.Scanner;`	`import java.util.Scanner;`
3	`public class HelloRabbit {`	`public class HelloRabbit {`
4	`  public static void main(String[] args){`	`  public static void main(String[] args){`
5	`      //make output`	`      //make output`
6	`     System.out.println("Don't worry!")`	`     System.out.println("Don't worry!");`
7	`     System.out.println("I can cure you")`	`     System.out.println("I can cure you");`
8	`  }`	`  }`
9	`}`	`}`

Figure 4.8. Mike’s code example of improvement

Figure 4.8 presents two continuous solutions of the student Mike in group 2. In the first solution, he missed the semicolons in line 6 and 7. Thus, this solution failed to be compiled, and the feedback telling him to add the semicolons was presented. In the next solution, this student added the necessary semicolons. While the **; expected** error was fixed, this solution still had compilation errors. In the comment in line 1, Mike wrote *Saymore*, which was the required program name for solving the problem **Say More**. However, in line 3, he named the class as *HelloRabbit*, which led to the **program name error**. In this study, this solution was labeled *not improved*, because the first (Solution #1) and the next (Solution #2) solution both had compilation errors. While Mike did improve his program and fixed the **; expected** error, the quantitative analysis did not identify the solution as improved. Therefore, the feedback message was effective in addressing the specific error, and so this feedback message showed positive effects even though the quantitative analysis did not detect it.

Test Error Feedback Message with Best Improvement Rate

The feedback message with the best improvement rate for addressing test errors was TFB4. It was designed to address the test error **TE4: Integer division issue** of the problem **Area of Triangle**. This error occurred when *int* type variables or values were used inappropriately in an expression with the division operator, because integer division in Java returns an *int* type value and ignores the decimal places. The feedback TFB4 explained how this error happened and provided a possible way to fix it. The analysis of student code indicated that students of group 2 made fewer intermediate solutions to fix this error.

#	Solution #1	Solution #2
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2		
3	`public class AreaOfTriangle {`	`public class AreaOfTriangle {`
4	`  public static void main(String[ ] args) {`	`  public static void main(String[ ] args) {`
5	`    Scanner in = new Scanner( System.in);`	`    Scanner in = new Scanner( System.in);`
6	`    int a = in.nextInt( );`	`    int a = in.nextInt( );`
7	`    int b = in.nextInt( );`	`    int b = in.nextInt( );`
8	`    int c = in.nextInt( );`	`    int c = in.nextInt( );`
9		
10	🟥 **`int`** `s = (a+b+c) /` **`2;`**	🟩 **`double`** `s = (a+b+c) /` **`2.0;`**
11		
12	`    a = (s - a);`	🟥 `a = (s - a);`
13	`    b = (s - b);`	🟥 `b = (s - b);`
14	`    c = (s - c);`	🟥 `c = (s - c);`
15		
16	`    double d = Math.sqrt(s*a*b*c);`	`    double d = Math.sqrt(s*a*b*c);`
17	`    String result=String.format("%.2f",d);`	`    String result=String.format("%.2f",d);`
18	`    System.out.println(result);`	`    System.out.println(result);`
19	`  }`	`  }`
20	`}`	`}`

Figure 4.9a. Group 1 student code example of improvement

According to the quantitative data, students of group 2 made this error six times, and five of them were successfully improved with the feedback. The analysis of the one failed case showed that the student also fixed this error, but the fix of the error led to another problem. Figure 4.9a shows the student's two solutions. The first solution had the integer division issue (see line 10). The second solution fixed this error but created a compilation error **possible loss of precision**, because the three assignment expressions in line 12, 13, and 14 all tried to assign double values to *int* type variables. Thus, this was not considered as an *improved* solution in the quantitative analysis, even though the error for which the feedback was given was successfully fixed.

#	Solution #1	Solution #2
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2	`public class AreaOfTriangle{`	`public class AreaOfTriangle{`
3	`  public static void main(String[] args) {`	`  public static void main(String[] args) {`
4	`  Scanner in = new Scanner(System.in);`	`  Scanner in = new Scanner(System.in);`
5	`  int a = in.nextInt();`	`  int a = in.nextInt();`
6	`  int b = in.nextInt();`	`  int b = in.nextInt();`
7	`  int c = in.nextInt();`	`  int c = in.nextInt();`
8	`  int sum = a + b + c;`	`  int sum = a + b + c;`
9	`  int s = sum / 2;`	`  double s = sum / 2;`
10	` //Student code were hidden`	` //Student code were hidden`
11	`  }`	`  }`
12	`}`	`}`

#	Solution #3	
1	`import java.util.Scanner;`	
2	`public class AreaOfTriangle{`	
3	`  public static void main(String[] args) {`	
4	`  Scanner in = new Scanner(System.in);`	
5	`  int a = in.nextInt();`	
6	`  int b = in.nextInt();`	
7	`  int c = in.nextInt();`	
8	`  int sum = a + b + c;`	
9	`  double s = sum / 2.0;`	
10	` //Student code were hidden`	
11	`  }`	
12	`}`	

Figure 4.9b. Emily's code example of improvement

When students in group 1 tried to fix this error, they tended to have more middle solutions. Figure 4.9b shows an example. This student, Emily, made this error in the first solution. In the next solution, she changed the type of the variable s from *int* to double. She was on the right track, but this change did not completely fix this error, because the division expression *sum / 2* would still return an integer value and ignore the decimal places. Finally, she fixed the error completely in the third solution. If she had received the feedback message, she might have fixed the error in Solution #2, instead of Solution #3.

Test Error Feedback Message with Worst Improvement Rate

The feedback message with the worst improvement rate for addressing test errors was TFB1. It was for addressing the test error **TE1: Mismatched input** of the problem **Area of Circle**. In this problem, the radius of the circle could be an integer or a decimal number (e.g., 5.9). When a student solution used the *nextInt()* method of the *Scanner* to read the radius, the mismatched input error occurred. Both groups had poor improvement rates on this error. The feedback TFB1 explained how the error occurred and offered code for fixing it. The analysis of student code found that students in group 2 had a better improvement rate than was shown in the quantitative analysis.

#	Solution #1	Solution #2
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2	`public class AreaOfCircle {`	`public class AreaOfCircle {`
3	`  public static void main (String[]args){`	`  public static void main (String[]args){`
4	`    Scanner in = new Scanner(System.in);`	`    Scanner in = new Scanner(System.in);`
5	`    int a = in.nextInt();`	`    double a = in.nextDouble();`
6	`    double area = a*a*3.14;`	`    double area = a*a*3.14;`
7		
8	`  //output`	`  //output`
9	`  String result=String.format("%.2f",area);`	`  String result=String.format("%.2f",area);`
10	`  System.out.println(area);`	`  System.out.println(area);`
11	` }`	` }`
12	`}`	`}`

Figure 4.10a. Alan's code example of improvement

According to the quantitative data, students of group 2 made this error six times, but only one of them successfully improved with the feedback. The analysis of student code showed that among the five "not improved" solutions, four were actually "improved," and fixed this error but still had other errors. Figure 4.10a presents such an example. This student, Alan, had the mismatched input error in Solution #1 and fixed this error in Solution #2. However, his second solution output the wrong variable; he should have printed the variable *result* rather than the variable *area*. In fact, this made Solution #2 get the test error **TE2: Wrong decimal places**. In this scenario, the quantitative analysis considered the solution as "not improved" even though the student was able to fix the identified error.

In contrast, students of group 1 made this error 28 times with three successful improvements. Among the other 25 failed cases, only three identified the error immediately and made some partial improvements. Figure 4.10b shows the improvement case of the student Mark. He made this error in Solution #1 and partially fixed it in Solution #2 by changing the type of the variable *a* to *double*. Meanwhile, he added a new line *String.format("%.2f", result)*. However, he forgot to add the semicolon to end this statement, which led to the compilation error **; expected**. Next, he fixed the **; expected** error in Solution #3 by adding the semicolon. Finally, in Solution #4, he fixed the other part of the mismatched input error, which was changing *nextInt()* method into *nextDouble()* method. While Alan fixed this error in the end, he required several steps. If the feedback message had been presented, he might not have required those intermediate solutions.

#	**Solution #1**	**Solution #2**
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2	`public class AreaOfCircle{`	`public class AreaOfCircle{`
3	`  public static void main(String[] args) {`	`  public static void main(String[] args) {`
4	`    Scanner in = new Scanner(System.in);`	`    Scanner in = new Scanner(System.in);`
5	`    int r = in.nextInt();`	`    double r = in.nextInt();`
6	`    double result = r * r * 3.14;`	`    double result = r * r * 3.14;`
7		`    String.format("%.2f", result)`
8	`    System.out.println(result);`	`    System.out.println(result);`
9	`  }`	`  }`
10	`}`	`}`

#	**Solution #3**	**Solution #4**
1	`import java.util.Scanner;`	`import java.util.Scanner;`
2	`public class AreaOfCircle{`	`public class AreaOfCircle{`
3	`  public static void main(String[] args) {`	`  public static void main(String[] args) {`
4	`    Scanner in = new Scanner(System.in);`	`    Scanner in = new Scanner(System.in);`
5	`    double r = in.nextInt();`	`    double r = in.nextDouble();`
6	`    double result = r * r * 3.14;`	`    double result = r * r * 3.14;`
7	`    String.format("%.2f", result);`	`    String.format("%.2f", result);`
8	`    System.out.println(result);`	`    System.out.println(result);`
10	`  }`	`  }`
11	`}`	`}`

Figure 4.10b. Mark's code example of improvement

Summary of Results

In this study, students' erroneous solutions were analyzed to identify common errors and specific misconceptions to address RQ 1. Fifty-five distinct compilation errors were identified, and 15 of them were categorized as common ones. The data also revealed that the 15 common compilation errors accounted for 92% of all compilation errors. Based on the 15 common compilation errors, three underlying student misconceptions were identified, including deficient knowledge of fundamental Java program structure, misunderstandings of Java expressions, and confusion about Java variables. In addition, 10 common test errors were identified based on nine difficult problems. The results showed that 54% of all test errors were related to those difficult problems. The 10 common test errors accounted for 39% of all test errors of the difficult problems. Four common student misconceptions were identified based on the 10 common test errors, including misunderstandings of Java input, misunderstandings of Java output, confusion about Java operators, and forgetting to consider special cases.

To address RQ2, both quantitative and qualitative data analysis were conducted to see whether and how the targeted feedback made a difference in students' solutions and so may have contributed to conceptual change. The results of quantitative analysis indicated that targeted feedback messages enhanced students' rates of improving erroneous solutions. Students of group 2 (the group receiving targeted feedback messages) showed significantly higher improvement rates in all erroneous solutions and solutions with common errors compared to students of group 1. Within group 2, students also showed a significantly higher improvement rate in solutions with targeted feedback messages compared to solutions without targeted feedback messages. All these results suggest that with targeted feedback messages, students were more likely to correct errors in their code. The qualitative analysis of students' solutions of four selected cases noted that when improving the code, students of group 2 made fewer intermediate incorrect solutions than students in group 1. In other words, the targeted feedback messages appear to have helped to promote conceptual change.

CHAPTER 5: DISCUSSION AND CONCLUSIONS

Student Misconceptions in Introductory Programming

Common Compilation Errors and Underlying Misconceptions

In this study, 55 distinct compilation errors were identified, and 15 of them were categorized as common ones. The results are consistent with previous studies on college students' compilation errors in introductory programming (see Becker, 2016 and Pettit et al., 2017). Most common compilation errors in this study were also found to be common among college CS1 students. However, there is one exception. The CE3: **program name error** found in this study did not appear on the common compilation error list of prior studies (Becker, 2016). This minor difference is not so surprising, because this error may not occur in a different instructional or research setting. For example, in a study of using the tool CodeWrite, students were requried to implement a *method* body to complete an exercise (Denny, Luxton-Reilly, & Tempero, 2012). In that study, students had no chance to make the **program name error**, as they did not have to write the program name. In addition, when the programming exercises allow arbitrary program names, the **program name error** should not be a common error because no specific program name is required. At the same time, students in this study did not have to define a *method* with a *return* statement, so errors related to the *return* statement, which were identified as common errors by prior studies (Brown & Altadmri, 2017; Denny et al., 2012), did not appear in this study. While minor differences exist between common compilation errors of this study and previous studies on college students, overall the common errors are similar. In other words, secondary school students in this study made similar common Java compilation errors to college students.

Based on the 15 common compilation errors, three underlying student misconceptions were identified, including deficient knowledge of fundamental Java program structure, misunderstandings of Java expressions, and confusion about Java variables. The first misconception, deficient knowledge of fundamental Java program structure is related to students' knowledge of basic Java syntax. In this study, students often made syntax errors, such as missing semicolons (CE2), incorrectly naming their programs (CE3), and mismatching braces and parentheses (CE4, CE5, CE7, CE9, and CE14). These errors seem to be superficial and trivial. Knowing relevant syntactic knowledge such as adding a semicolon to end a statement is not challenging, and many times students in this study were able to construct syntactically correct programs or fix those syntax errors eventually. This suggests that students have knowledge about Java program structure. However, the repetition of these syntax errors indicates that there may exist a deeper problem. While students may be aware of relevant syntactic knowledge about Java program structure, they may not be able to understand and apply the knowledge correctly (Krathwohl, 2002). When the task gets complicated, the task complexity and students' unfamiliarity with Java syntax may increase the demand on cognitive load so that students may have difficulties (Sanders & Thomas, 2007; Sweller, 1988). Hence, they start to omit semicolons, use wrong program names, and mismatch braces. Taber (2013) points out that conceptual knowledge (knowledge of concepts) has implicit elements that are not typically taught in class. For example, students in this study were taught the fundamental Java program structure, but they did not learn why the semicolons are necessary, the basic mechanism of the compiler, and what possible error messages would be when the program structure is wrong. Without the implicit knowledge, students may be able to write a correct Java program, but they have deficient knowledge that may lead them to make relevant mistakes. Students may learn certain implicit knowledge during the programming practice by themselves. However, explicitly teaching implicit knowledge of Java program structure, instead of simply introducing the facts about the fundamental Java program structure, may help students better understand the concept (Muller et al., 2007; Sajaniemi & Kuittinen, 2005).

The other two misconceptions, misunderstandings of Java expressions and confusion about Java variables, are mainly related to the conflicts between students' existing knowledge and new knowledge. While students' unfamiliarity of Java syntax may contribute to their errors of using Java expressions and variables, the major interference appears to be from students' prior knowledge. While sometimes students' errors in constructing Java expressions were due to certain syntax problems (e.g. missing or mismatching quotation marks), the analysis of student code revealed that students frequently attempted to write Java expressions in ways similar to how they would write expressions in math class. For instance, some students omitted the multiplication operator (*) in their expressions (e.g. $d = a + 0.1a + 0.01a$). While the multiplication operator can be omitted in a math expression, it is required in a Java expression.

Similarly, students' prior knowledge about the concept variable also interfered with their learning of Java variables and variable operations. For example, many students in this study made errors related to variable types and precision. Variables in math do not have a specific type and have unlimited precision, but in Java variable types and precision must be specified.

Previous research has indicated that one dominant source of students' misconceptions is their prior knowledge (Bonar & Soloway, 1985; Smith et al., 1994). In the learning of computer programming, students' existing math knowledge is an important factor contributing to student misconceptions (Clancy, 2004; Qian & Lehman, 2017). Students construct new knowledge based on their existing knowledge (Ausubel, 2000; Jonassen, 1991), and when the new knowledge conflicts with their prior knowledge, students have confusion between pre-instructional conceptions and new conceptions and thus misconceptions begin to form (Özdemir & Clark, 2007; Taber, 2013). Java expressions and variables share many similarities with those of math. When students in this study wrote their Java code, they sometimes confused Java expressions and variables with the math ones, as they had learned similar concepts in math. According to conceptual change theories (Özdemir & Clark, 2007; Taber, 2013), a student's misconception has both correct and incorrect elements. To promote conceptual change, it is important to help students fix the incorrect elements and refine their understanding of the relationships between new knowledge and existing conceptual structure (diSessa, 2014). Hence, in instruction, teachers should highlight the differences between Java and math knowledge to help students reduce confusion and reach a better understanding of the computer science concepts.

It is important to note that all the compilation errors could be detected by the IDE students used, drJava, on their local machine. In other words, if students had compiled their solutions before submitting them to Mulberry, they should not have had any compilation errors in their code. However, students in this study still made a large number of compilation errors. This may indicate that students often failed to use the IDE to check if their solutions had compilation errors before submitting them to Mulberry. When student developed their solutions, they often only used the built-in Java editor of Mulberry (see Figure 3.1) and failed to use drjava to test their solutions. Another possibility is that students could not understand the compiler error messages given by the IDE. For beginners, raw compiler error messages are "cryptic and uninformative, often terse and misleading" (Becker, 2016, p. 126). Compiler error messages usually describe errors using technical terms, which make them difficult to understand. In addition, these error messages are not always precise and sometimes point to the lines with no errors. Therefore, it is possible that students tried to compile their solutions using the IDE, but they could not understand the error messages from the IDE and so submitted their solutions as is ignoring the errors. Prior studies have indicated that novices often have limited knowledge of compiler error messages, locating errors, and fixing errors (Becker, 2016; Fitzgerald et al., 2008; McCauley et al., 2008). Hence, explicitly teaching knowledge of compiler error messages and skills of debugging may help students better use the features of the IDE and reduce the compilation errors in their programs.

Common Test Errors and Underlying Misconceptions

In this study, 10 common test errors were identified. Different from compilation errors, test errors are problem-specific. While previous studies have used student data in automated assessment systems to identify common compilation errors, few of them have examined common test errors and relevant student misconceptions. In this study, the 10 common test errors were identified based on nine difficult problems. Our results showed that 54% of all test errors were related to those difficult problems. The 10 common test errors accounted for 39% of all test errors of the difficult problems. In other words, difficult problems and common test errors can play an important role in understanding student misconceptions. Hence, researchers and educators should pay attention to students' non-compilation errors, rather than only focusing on the compilation errors.

As common tests errors are based on specific programming problems, the error details themselves may not be important to educators and researchers in other instructional settings. However, the relevant student misconceptions behind the errors can be meaningful and helpful to others. Four common student misconceptions were identified based on the 10 common test errors, including misunderstandings of Java input, misunderstandings of Java output, confusion about Java operators, and forgetting to consider special cases.

The first two misconceptions, misunderstandings of Java input and output, are related to the concept Input/Output (I/O). In computer programming, the required input and expected output of a program must be exact. If a program can only read one integer at one time, inputting two integers or decimal numbers will result in errors and/or make the program crash. Similarly, if the expected output of a program is two words separated by a comma, outputting two words separated by a space or without a separator makes this program inaccurate or incorrect. However, students in this study may not have understood the need for exactness of I/O in computer programming. They often failed to design code to read input or produce output in an exact way. One problem in Mulberry required a three-digit integer as input, but many students designed a program that read three integers as three digits rather than a single three-digit integer. Several problems required outputting decimal numbers with only two decimal places; however, students repeatedly failed to keep two decimal places. As novices, they might have had difficulties in using Java statements related to I/O. However, I/O statements were used in almost every program they wrote, and code examples for reading various kinds of inputs and producing special output (e.g. outputting a number with n decimal places) were introduced in class and/or provided in problem descriptions. Hence, students' unfamiliarity with relevant Java statements may not be an essential problem, but their misunderstandings of the required exactness of I/O in computer programming can be vital.

First, students' everyday experience of using computers may make them believe that computers have certain intelligence like a human and can understand what they mean (Miller, 2014; Pea, 1986). These days, students use graphical/touchable user interfaces all the time and may have limited experience in using a text-based user interface. Intelligent technologies such as Apple Siri and Microsoft Cortana make computers more human-like. Such life experiences may bring them a feeling that computers are smart enough to understand what they mean in the code. Second, their natural language may also make them believe that vagueness in code does not matter (Bonar & Soloway, 1985; Miller, 2014). Therefore, in instruction, teachers should help students build the understanding that computers are machines precisely executing code line by line and have no ability to understand ambiguous code.

The third misconception, confusion about Java operators, is related to students' prior math knowledge. In this study, two operators, the division operator (/) and the equality operator (==), were frequently used incorrectly by students. While the Java equality operator does not exist in math, students had learned the equality concept and meaning of the equals sign (=) in math. In math, when using the division operator to check for equality of two values, the types of the operands will not affect the results. For example, the mathematical expressions *11/2* and *11/2.0* both give 5.5. However, in Java, when the types of the operands are different, the result of the operation may be different depending on the operator or method used (e.g. checking for equality of *String* values). When students in this study used Java operators, they might not have appreciated the differences, even though the differences were introduced in class. On the other hand, the differences between Java and math operators are subtle and difficult for students to notice.

Inappropriate use of operators typically does not produce any compilation errors. Without an obvious error message, students may believe that no error exists in their code (Ben-David Kolikant & Mussai, 2008). In addition, students need sufficient knowledge about Java variables to fully understand why an operator has certain behaviors. For example, to understand why the equality operator cannot be used to compare *double* or *String* type variables, students have to know that floating-point numbers are not precise in programming, and *Strings* are different from primitive data types. Therefore, in instruction, only introducing special features of certain Java operators may not help students understand the differences between Java and math operators. Explaining operator-related concepts such as variables and providing details about relationships between Java operators and variables may be an effective instructional strategy.

The last misconception, forgetting to consider special cases, is related to students' strategic programming knowledge (also called programming strategies). For novices who have just learned a programming language, their knowledge of the syntax and programming concepts is usually fragmentary and not well-organized into meaningful structures (Clancy & Linn, 1999; Davies 1993; Lister et al., 2006;). Novices may be able to write a syntactically correct program but fail to consider boundaries of conditions and unexpected cases, because they lack certain patterns and strategies that experts use to evaluate and debug programs (Fisler et al., 2016; Sajaniemi & Prieto, 2005). In this study, students were all novices, so it is not surprising that they forgot to inspect boundary conditions and/or special cases. Prior studies have noted that explicitly teaching programming strategies, such as debugging strategies, can improve students' strategic knowledge and help them better understand programming concepts (Muller et al., 2007; Qian &Lehman 2017; Sajaniemi & Kuittinen, 2005).

Feedback for Conceptual Change

Overall Effects of Feedback

The results of this study indicated that targeted feedback messages enhanced students' improvement rates of erroneous solutions. Students of group 2 (the group receiving targeted feedback messages) showed significantly higher improvement rates in all erroneous solutions and solutions with common errors than students of group 1. Within group 2, students also showed a significantly higher improvement rate in solutions with targeted feedback messages compared to solutions without targeted feedback messages. All these results suggest that with targeted feedback messages, students were more likely to correct errors in their code. This finding is consistent with previous research (Becker et al., 2016).

In the study of Becker et al. (2016), researchers provided feedback for 30 common compilation errors by enhancing the compiler error messages. They found that the 30 compilation errors accounted for 78% of all errors, and the group receiving feedback messages made 32% fewer errors than the group only seeing the original Java compiler error messages. While the study of Becker et al. (2016) only investigated students' compilation errors, its overall research approach and results are similar to this study. In this study, targeted feedback messages were designed and provided for both common compilation and test errors. The data of this study revealed that the 15 common compilation errors accounted for 92% of all compilation errors. For the test errors, 54% of them were related to those difficult problems. The 10 common test errors accounted for 39% of all test errors of the difficult problems. Therefore, one important step of designing the feedback component of an automated assessment system is to identify the common errors students make, which are the representatives of common difficulties students encounter in learning to programming.

In contrast, two recent studies reported conflicting results (Denny et al., 2014; Pettit et al., 2017). Both studies examined the effects of enhanced compiler error messages and indicated that no significant effects were found. However, the research design of the two studies was different from Becker et al.'s (2016) and this study and may account for the lack of significant results. In the study of Denny et al. (2014), students only had to complete the method body of a given method header. Hence, students did not have to write a program from scratch and would not encounter all possible Java compilation errors (Becker et al., 2016). The study of Pettit et al.

(2017) also indicated that enhanced compiler error messages did not benefit students. However, their feedback messages only covered 30% of compilation errors, which may make the effects of their feedback insignificant. While many factors may contribute to the ineffectiveness of feedback in the two studies, one key issue is that the feedback messages they offered may not have addressed the common student errors in their instructional settings. Without a good identification of common student errors that can account for most student errors, the feedback system of an automated assessment system may not work as expected. As the visibility model of feedback suggests, designing effective feedback requires precisely analyzing and understanding learners' current knowledge states (Hattie & Gan, 2011; Hattie & Timperley, 2007). If the design of feedback messages is not based on students' current knowledge states and only addresses a limited number of student errors, the feedback may not be as effective as expected.

Evolution of Students' Misconceptions

The qualitative analysis of students' solutions of four selected cases noted that when improving the code, students of group 2 made fewer intermediate incorrect solutions than students in group 1. In other words, the targeted feedback messages appear to have helped to promote conceptual change. According to the qualitative analysis, students of group 1 usually noticed the error but often only fixed part of the error in the next attempted solution (see Figure 4.9b for an example). In contrast, with the targeted feedback message, students of group 2 often could completely fix the same error in the revised solution (see Figure 4.9a for an example). According to conceptual change theories (Taber, 2013; Vosniadou, 1994; Vosniadou & Skopeliti, 2014), before students develop correct understanding of an academic concept, they may gain certain intermediate states of knowledge because of the conflicts and interactions between their existing knowledge and the new concept. Such intermediate states of knowledge are called melded concepts (Taber, 2013) or synthetic models (Vosniadou & Skopeliti, 2014) and consist of both correct and erroneous knowledge elements (diSessa, 2014). From this viewpoint, both students of group 1 and 2 formed certain melded concepts, but the targeted feedback messages might have helped students of group 2 correct the erroneous knowledge elements and reduce the intermediate states. Conceptual change is an evolutionary process of correcting and enhancing existing knowledge elements and establishing and refining the relationships among conceptions (Abimbola, 1988; diSessa, 2013). Therefore, when providing feedback for students, it is important to analyze students' possible melded concepts and consider their (mis)conceptions as resources, rather than trying to replace them (Smith et al., 1994).

In addition, the qualitative analysis revealed that quantitative analysis in this study failed to detect certain improvements in student code, and the targeted feedback messages might have worked better than the quantitative results suggested. When analyzing the two feedback message cases with the worst improvement rates, the results showed that the quantitative analysis labeled some solutions as "not improved" even though the error related to the feedback was fixed because other errors were still present. This is a limitation of quantitative analysis (Fields et al., 2016) and also highlights the value of qualitative analysis of student code. Therefore, it is important to find new techniques or algorithms to improve the accuracy of the quantitative analysis, because manually conducting qualitative analysis of every student solution is time-consuming and difficult.

Implications

An effective CS teacher needs to have both knowledge of the subject matter and pedagogical content knowledge (Hubwieser, Magenheim, Mühling, & Ruf, 2013; Shulman, 1986; Yadav, Berges, Sands, & Good, 2016). Pedagogical content knowledge (PCK) refers to the knowledge that enables teachers to transform instructional content into a comprehensible form to students (Shulman, 1986). One key component of teachers' PCK is their knowledge of students' misconceptions (Carlsen, 1999; Saeli, Perrenet, Jochems, & Zwaneveld, 2011; Shulman, 1986). Unfortunately, research on CS teachers' PCK is limited (Saeli et al., 2011; Yadav et al., 2016), and CS teachers often lack sufficient understanding of student misconceptions (Brown & Altadmri, 2017; Guzdial, 2015). CS teachers' teaching and computing experience sometimes lead them to form "misconceptions" of common student misconceptions. Further, as experts, they sometimes ignore the difficulties novice students have (Brown & Altadmri, 2017; Guzdial, 2015). The results of this study suggest a data-driven approach to understanding and addressing student misconceptions, which is using student data in automated assessment systems, has the potential to help teachers build a more accurate understanding of their students' common misconceptions and develop their PCK.

Automated assessment systems have been widely used in college-level introductory programming classes (Douce et al., 2005; Pettit et al., 2017). They can not only reduce teachers' grading workload but also collect a large amount of student data including all errors students make. The student data in automated assessment systems can be a good resource for analyzing student misconceptions. Previous studies have used such data and catalogued a broad range of misconceptions of college students (Altadmri & Brown, 2015; Becker, 2016; Denny et al., 2012). However, college CS1 instructors often have limited knowledge of students' misconceptions (Brown & Altadmri, 2017; Spohrer & Soloway, 1986). Hence, it is important for college CS1 instructors to utilize the data in automated assessment systems to understand their students' common misconceptions. Researchers and developers have already developed a variety of tools that can identify common student misconceptions using the data in automated assessment systems (Becker, 2016; Denny et al., 2012). Instead of creating new tools, college CS1 instructors should learn to use existing tools. Meanwhile, researchers and developers should keep improving their tools. For example, many existing tools do not consider students' non-compilation errors, but the results of this study show that non-compilation errors are also important to understand student misconceptions. Therefore, researchers and developers of automated assessment systems should develop components that support identifying common student misconceptions using both compilation and non-compilation errors.

As CS education has been expanding into K-12 schools, CS teachers at the pre-college level should also learn to use automated assessment systems and student data to understand student misconceptions. Because students' misconceptions are contextually sensitive (diSessa, 2013; Özdemir & Clark, 2007), pre-college students in different instructional settings may show different misconceptions. While the results of this study indicate that secondary school students make similar common errors to those college students, minor difference still existed, such as the program name error. Thus, integrating automated assessment systems with misconception identification components into pre-college introductory programming courses can be helpful and valuable. However, such systems may not always accessible to CS teachers in K-12 schools. Some other approaches may also be useful to help teachers understand student misconceptions in introductory programming. For example, a professional development program with a focus on common student misconceptions may benefit teachers who have limited knowledge of student misconceptions (Qian, Hambrusch, Yadav, & Gretter, 2018). The development and use of a concept inventory is another potential way to help teachers evaluate students' understanding of fundamental programming concepts (Goldman et al., 2010; Taylor et al., 2014).

In addition to identifying student misconceptions, teachers also need to have the ability to address misconceptions. Adding a well-designed feedback system to an automated assessment system can be one good solution. Some researchers have developed and studied the systems that can offer automatically generated feedback using artificial intelligence techniques (Price et al., 2017; Rivers & Koedinger, 2017). While such systems seem to be an ideal solution to help teachers address student misconceptions, they are yet not mature and can only handle simple programs. According to the results of this study and previous research, the key to provide effective feedback for addressing student misconceptions is an accurate understanding of students' common misconceptions. While students may make a variety of errors in their code, a small number of common errors account for most student errors. Hence, no matter whether a feedback system exists, teachers should focus on common student errors and difficult problems identified by the student data. The ability to effectively identify and address common misconceptions based on student data will be vital to quality CS teachers.

Limitations and Future Research Directions

While plausible results were found, this study has several limitations. First, the generalizability of findings from this study is limited. In this study, participants were high-ability students. Their misconceptions may not be representative of the population of secondary school students. Further, because the students were participating in a non-school-based summer enrichment program that was not formally graded, students may not have been motivated to learn. Hence, future research on ordinary middle and high school students in formal educational settings is necessary to better understand secondary school students' common misconceptions. In addition, as two Advanced Placement (AP) CS courses, AP Computer Science Principles and AP Computer Science A, have been developed for high-ability high school students, conducting research on students who take the AP CS courses in formal educational settings is important to better understand common misconceptions of high-ability students.

In addition, the sample size of this study was relatively small. In this study, group 1 and 2 had only 13 and 10 students respectively. Previous studies on college students often had a sample of more than 100 students (e.g. Becker, 2016). While this study included students from summer 2016 to obtain a more complete understanding of common student misconceptions, the analysis of the effects of feedback was limited to the data of the 23 students of summer 2017. Because of the small sample size, certain feedback messages were only triggered a limited number of times. For example, CFB5 and TFB7 were only triggered once in this study (see Table 4.4 and 4.5). Thus, with such limited data, the effects of such feedback messages could not be determined. Future research should use a larger sample to better examine the effects of feedback.

Another limitation is there was no control group in this study. Although the use of a control group is not typical in design-based studies, without a control group it is not possible to establish a causal relationship between the observed changes and the intervention. While the results of this study suggest that with targeted feedback messages students are more likely to correct errors in their code, without a strictly controlled experiment it is difficult to determine whether the feedback causes the improvements. Students of the two groups might have differences in prior knowledge of programming, existing math knowledge, and other factors. It is possible that there existed certain confounding variables that resulted in the higher improvement rates of group 2 students, rather than the feedback. Thus, to further investigate the effects of feedback, it is essential to implement a study with both treatment and control groups by controlling variables such as students' gender, previous computing experience, academic performance in other subjects, and so forth.

The programming problems in Mulberry are also a limitation because they may not reveal all possible student misconceptions. For example, no problems in Mulberry were related to the concepts Classes and Objects, which have been shown to be problematic for beginners in previous studies. Therefore, it is important to expand the problem pool of Mulberry to cover a broader range of possible concepts that are included in a typical introductory Java programming course. Further, when Mulberry has more problems that cover more programming concepts, the course design should be revised and expanded. The course in this study was designed for a two-week summer camp and only covered a limited number of programming concepts. During the two weeks, most students could only solve a small number of problems in Mulberry. Certain common misconceptions may have remained hidden, even though relevant problems were included in Mulberry. For example, the concept Loops has been shown to be difficult for novices, but no difficult problems in this study were related to Loops, because only a few students in this study solved problems related to Loops. Hence, both the problem pool of Mulberry and the course time should be expanded to cover more programming concepts so that future research may be able to reveal additional student misconceptions.

Another limitation of this study is that the categorization of “improved” and “not improved” student solutions may not accurately reflect student performance. According to the qualitative analysis of student programs, certain improvements in students’ code, such as fixing a common error, were not detected by the categorization algorithm used in this study. Hence, future research is needed to see if a more accurate categorization algorithm can be developed. For example, machine learning techniques might be implemented to analyze student code directly to identify improvements.

Finally, the results of this study also provide potential directions for future studies on student misconceptions in introductory programming. One key feature of this study was the use of students’ non-compilation errors. The results indicate the importance and value of these non-compilation errors in understanding student misconceptions. Hence, future research should pay more attention to students’ non-compilation errors and examine possible ways to reveal and analyze various kinds of non-compilation errors, such as logic errors, run-time errors, and so on. Furthermore, many factors may affect the effectiveness of feedback, such as students’ confidence, the difficulty of the problem, time of reading the feedback, and so on. Future research should investigate the impacts of these variables on the effectiveness of the feedback. As natural language may interfere with the learning of programming, future research should examine common misconceptions related to language. For example, the performance of non-English speakers might be compared to native English speakers to see whether they show the same misconceptions and how their English ability impacts their learning of programming.

Conclusions

With the expansion of computer science education, CS teachers in K-12 schools should be cognizant of student misconceptions and be prepared to help students establish accurate understanding of computer science and programming. This exploratory design-based research study implemented a data-driven approach to identify secondary school students’ misconceptions and provide targeted feedback to promote students’ conceptual change in introductory programming. A total of 15 common compilation errors and 10 common test errors were identified in this study. The results showed that these common errors accounted for a large proportion of all errors. The results suggest that identifying common errors, both compilation and test errors, is important to teach introductory programming courses. Based on these common errors, seven underlying student misconceptions were identified. A variety of factors that may contribute to the misconceptions were discussed, including students' deficient programming knowledge, prior math knowledge, everyday experience, and lack of strategic knowledge. Possible instructional strategies to address these student misconceptions were also discussed.

Based on students' common errors and underlying misconceptions, targeted feedback messages were designed and provided for students. The quantitative analysis found that with targeted feedback students were more likely to correct the errors in their code. The qualitative analysis of students' solutions revealed that when improving the code, students receiving feedback made fewer intermediate incorrect solutions. In other words, the targeted feedback messages may help to promote conceptual change. The results suggest that designing effective feedback for promoting conceptual change requires precisely analyzing students' current (mis)conceptions. Hence, this study proposes a data-driven approach to understand and address student misconceptions, which is using student data in automated assessment systems, to both improve student learning of programming and help teachers build accurate understanding of their students' common misconceptions and develop their PCK. Researchers and developers of automated assessment systems should develop components that support identifying common student misconceptions using both compilation and non-compilation errors. No matter whether the automated assessment system has a feedback system, teachers should utilize common student errors and difficult problems identified by the student data to support and improve their instruction.

These days, computer science education is expanding quickly, but research on CS teachers' PCK is limited. Digital tools, such as automated assessment systems, definitely can be useful and supportive in teaching CS courses. The findings of this exploratory study showed evidence of the power of digital tools. However, more research is needed to make technology benefit more CS teachers. One issue of this study is that the current quantitative analysis method used in this study may miss certain improvements in students' code. The next step of research should focus on finding more accurate analysis methods to analyze students' improvements in coding.

REFERENCES

Abimbola, I. O. (1988). The problem of terminology in the study of student conceptions in science. *Science Education*, *72*(2), 175–184. https://doi.org/10.1002/sce.3730720206

Altadmri, A., & Brown, N. C. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522–527). New York, New York, USA: ACM Press. https://doi.org/10.1145/2676723.2677258

Anderson, J. R., Boyle, C. F., & Reiser, B. J. (1985). Intelligent tutoring systems. *Science*, *228*(4698), 456–462. https://doi.org/10.1126/science.228.4698.456

Anderson, T., & Shattuck, J. (2012). Design-based research: A decade of progress in education research? *Educational Researcher*, *41*(1), 16–25. https://doi.org/10.3102/0013189X11428813

Ausubel, D. P. (2000). *The acquisition and retention of knowledge: A cognitive view*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Azevedo, R., & Bernard, R. M. (1995). A meta-analysis of the effects of feedback in computer-based instruction. *Journal of Educational Computing Research*, *13*(2), 111–127.

Balzer, W. K., Doherty, M. E., & O'Connor, R. (1989). Effects of cognitive feedback on performance. *Psychological Bulletin*. https://doi.org/10.1037/0033-2909.106.3.410

Bangert-Drowns, R. L., Kulik, C. L. C., Kulik, J. A., & Morgan, M. (1991). The instructional effect of feedback in test-like events. *Review of Educational Research*, *61*(2), 213–238. https://doi.org/10.3102/00346543061002213

Barnes, T., & Stamper, J. (2010). Automatic hint generation for logic proof tutoring using historical data. *Educational Technology and Society*, *13*(1), 3–12.

Bayman, P., & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology*, *80*(3), 291–298. https://doi.org/10.1037/0022-0663.80.3.291

Becker, B. A. (2016). An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16* (pp. 126–131). https://doi.org/10.1145/2839509.2844584

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, *26*(2–3), 148–175. https://doi.org/10.1080/08993408.2016.1225464

Bell, T., Andreae, P., & Robins, A. (2014). A case study of the introduction of computer science in NZ schools. *ACM Transactions on Computing Education*, *14*(2), 1–31. https://doi.org/10.1145/2602485

Ben-David Kolikant, Y., & Mussai, M. (2008). “So my program doesn’t run!” Definition, origins, and practical expressions of students’ (mis)conceptions of correctness. *Computer Science Education*, *18*(2), 135–151. https://doi.org/10.1080/08993400802156400

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, *1*(2), 133–161. https://doi.org/10.1207/s15327051hci0102_3

Brown, A. L. (1992). Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *The Journal of the Learning Sciences*, *2*(2), 141-178.

Brown, N. C. C., & Altadmri, A. (2017). Novice java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education*, *17*(2), 7:1--7:21. https://doi.org/10.1145/2994154

Brown, N. C. C., Sentance, S., Crick, T., & Humphreys, S. (2014). Restart: The resurgence of computer science in uk schools. *ACM Transactions on Computing Education*, *14*(2), 1–22. https://doi.org/10.1145/2602484

Bruckman, A., & Edwards, E. (1999). Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In ACM (Ed.), *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems- CHI ’99* (pp. 207–214). New York, NY, USA. https://doi.org/10.1145/302979.303040

Butler, A. C., Karpicke, J. D., & Roediger, H. L. (2007). The effect of type and timing of feedback on learning from multiple-choice tests. *Journal of Experimental Psychology: Applied*, *13*(4), 273–281. https://doi.org/10.1037/1076-898X.13.4.273

Butler, D. L., & Winne, P. H. (1995). Feedback and self-regulated learning: A theoretical synthesis. *Review of Educational Research*, *65*(3), 245–281. https://doi.org/10.3102/00346543065003245

Carlsen, W. (1999). Domains of teacher knowledge. In J. Gess-Newsome & N. G. Lederman (Eds.), *Examining pedagogical content knowledge: The construct and its implications for science education* (pp. 133–144). Kluwer Academic Publishers.

Clancy, M. J. (2004). Misconceptions and attitudes that interfere with learning to program. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 85–100). London, UK: Taylor & Francis Group.

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *ACM SIGCSE Bulletin*, *31*(1), 37–42. https://doi.org/10.1145/384266.299673

Clariana, R. B., Wagner, D., & Murphy, L. C. R. (2000). Applying a connectionist description of feedback timing. *Educational Technology Research and Development*, *48*(3), 5–22. https://doi.org/10.1007/BF02319855

Clement, J. (1993). Using bridging analogies and anchoring intuitions to deal with students' preconceptions in physics. *Journal of Research in Science Teaching*, *30*(10), 1241–1257. https://doi.org/10.1002/tea.3660301007

Corbett, A. T., & Anderson, J. R. (2001). Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '01* (pp. 245–252). New York, NY, USA: ACM Press. https://doi.org/10.1145/365024.365111

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 141–146). New York, NY, USA: ACM. https://doi.org/10.1145/2157136.2157180

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*(2), 237–267. https://doi.org/10.1006/imms.1993.1061

De-La-Fuente-Valentín, L., Pardo, A., & Delgado Kloos, C. (2013). Addressing drop-out and sustained effort issues with large practical groups using an automated delivery and assessment system. *Computers and Education*, *61*(1), 33–42. https://doi.org/10.1016/j.compedu.2012.09.004

de Raadt, M. (2008). *Teaching programming strategies explicitly to novice programmers*. Doctoral dissertation, University of Southern Queensland.

Dempsey, J. V., Driscoll, M. P., & Swindell, L. K. (1993). Text-based feedback. In J. V. Dempsey & G. C. Sales (Eds.), *Interactive Instruction and Feedback* (pp. 21–54). Englewood Cliffs, NJ: Educational Technology Publications.

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. *Proceedings of the 19th ACM Conference on Innovation & Technology in Computer Science Education*, 273–278. https://doi.org/10.1145/2591708.2591748

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012). All syntax errors are not equal. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE '12*, 75–80. https://doi.org/10.1145/2325296.2325318

desJardins, M. (2015). Creating AP® CS principles: Let many flowers bloom. *ACM Inroads*, *6*(4), 60–66. https://doi.org/10.1145/2835852

diSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and Instruction*, *10*(2–3), 105–225. https://doi.org/10.1080/07370008.1985.9649008

diSessa, A. A. (2013). A bird's-eye view of the "pieces" vs "coherence" controversy (from the "pieces" side of the fence). In S. Vosniadou (Ed.), *International Handbook of Research on Conceptual Change* (pp. 31–48). New York, NY: Taylor and Francis.

diSessa, A. A. (2014). The construction of causal schemes: Learning mechanisms at the knowledge level. *Cognitive Science*, *38*(5), 795–850. https://doi.org/10.1111/cogs.12131

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming. *Journal on Educational Resources in Computing*, *5*(3), 4:1-13. https://doi.org/10.1145/1163405.1163409

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Duit, R., & Treagust, D. F. (2003). Conceptual change: A powerful framework for improving science teaching and learning. *International Journal of Science Education*, *25*(6), 671–688. https://doi.org/10.1080/09500690305016

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human - Computer Studies*, *41*(4), 457–480. https://doi.org/10.1006/ijhc.1994.1069

Fields, D. A., Quirke, L., Amely, J., & Maughan, J. (2016). Combining big data and thick data analyses for understanding youth learning trajectories in a summer coding camp. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 150–155). New York, NY, USA: ACM. https://doi.org/10.1145/2839509.2844631

Fisler, K., Krishnamurthi, S., & Siegmund, J. (2016). Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16* (pp. 211–216). New York, New York, USA: ACM Press. https://doi.org/10.1145/2839509.2844556

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education*, *27*(1), 65–100. https://doi.org/10.1007/s40593-015-0080-x

GERI Website. (2018). GERI Summer Residential Program. Retrieved March 15, 2018, from https://www.education.purdue.edu/geri/youth-programs/summer-residential/

Gielen, S., Peeters, E., Dochy, F., Onghena, P., & Struyven, K. (2010). Improving the effectiveness of peer feedback for learning. *Learning and Instruction*, *20*(4), 304–315. https://doi.org/10.1016/j.learninstruc.2009.08.007

Gilman, D. A. (1969). Comparison of several feedback methods for correcting errors by computer-assisted instruction. *Journal of Educational Psychology*, *60*(6), 503–508. https://doi.org/10.1037/h0028501

Ginat, D., Shifroni, E., & Menashe, E. (2011). Transfer, cognitive load, and program design difficulties. In I. Kalaš & R. T. Mittermeir (Eds.), *Informatics in Schools. Contributing to 21st Century Education* (pp. 165–176). Berlin, Heidelberg: Springer Berlin Heidelberg.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the scope of concept inventories for introductory computing subjects. *ACM Transactions on Computing Education*, *10*(2), 1–29. https://doi.org/10.1145/1789934.1789935

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. *SIGCSE 2013 - Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 579–584. https://doi.org/10.1145/2445196.2445368

Guzdial, M. (1995). Centralized mindset: A student problem with object-oriented programming. *ACM SIGCSE Bulletin*, *27*(1), 182–185. https://doi.org/10.1145/199691.199772

Guzdial, M. (2015). Learner-centered design of computing education: research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, *8*(6), 1–165. https://doi.org/10.2200/S00684ED1V01Y201511HCI033

Hattie, J., & Gan, M. (2011). Instruction based on feedback. In R. E. Mayer & P. A. Alexander (Eds.), *Handbook of Research on Learning and Instruction* (pp. 249–271). New York, NY: Routledge.

Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of Educational Research*, *77*(1), 81–112. https://doi.org/10.3102/003465430298487

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin*, *29*(1), 131–134. https://doi.org/10.1145/268085.268132

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 153–156). New York, NY, USA: ACM. https://doi.org/10.1145/611892.611956

Hubwieser, P., Magenheim, J., Mühling, A., & Ruf, A. (2013). Towards a conceptualization of pedagogical content knowledge for computer science. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research - ICER '13*, 1–8. https://doi.org/10.1145/2493394.2493395

Jackson, J., Cobb, M., & Carver, C. (2005). Identifying top Java errors for novice programmers. In *Proceedings Frontiers in Education 35th Annual Conference* (p. T4C–T4C). https://doi.org/10.1109/FIE.2005.1611967

Jaehnig, W., & Miller, M. L. (2007). Feedback types in programmed instruction: A systematic review. *The Psychological Record*, *57*(2), 219–232. https://doi.org/10.1007/BF03395573

Jonassen, D. H. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development*, *39*(3), 5–14. https://doi.org/10.1007/BF02296434

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 107–111). New York, NY, USA: ACM. https://doi.org/10.1145/1734263.1734299

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), 83–137. https://doi.org/10.1145/1089733.1089734

Klopfer, L. E., Champagne, A. B., & Gunstone, R. F. (1983). Naive knowledge and science learning. *Research in Science & Technological Education*, *1*(2), 173–183. https://doi.org/10.1080/0263514830010205

Kluger, A. N., & DeNisi, A. (1996). The effects of feedback interventions on performance: A historical review, a meta-analysis, and a preliminary feedback intervention theory. *Psychological Bulletin*, *119*(2), 254–284.

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, *16*(1–2), 41–84. https://doi.org/10.1016/J.JVLC.2004.08.003

Kölling, M. (2010). The Greenfoot programming environment. *ACM Transactions on Computing Education*, *10*(4), 1–21. https://doi.org/10.1145/1868358.1868361

Krathwohl, D. R. (2002). A revision of bloom’s taxonomy: An overview. *Theory Into Practice*, *41*(4), 212–218. https://doi.org/10.1207/s15430421tip4104_2

Kulhavy, R. W. (1977). Feedback in written instruction. *Review of Educational Research*, *47*(2), 211–232. https://doi.org/10.3102/00346543047002211

Kulhavy, R. W., & Anderson, R. C. (1972). Delay-retention effect with multiple-choice tests. *Journal of Educational Psychology*, *63*(5), 505–512. https://doi.org/10.1037/h0033243

Kulhavy, R. W., & Stock, W. A. (1989). Feedback in written instruction: The place of response certitude. *Educational Psychology Review*, *1*(4), 279–308. https://doi.org/10.1007/BF01320096

Kulhavy, R. W., & Wager, W. (1993). Feedback in programmed instruction: Historical context and implications for practice. In J. V. Dempsey & G. C. Sales (Eds.), *Interactive Instruction and Feedback* (pp. 3–20). Englewood Cliffs, NJ: Educational Technology Publications.

Kulik, J. A., & Kulik, C. L. C. (1988). Timing of feedback and verbal learning. *Review of Educational Research*, *58*(1), 79–97. https://doi.org/10.3102/00346543058001079

Lee, O. M. (1985). *The effect of type of feedback on rule learning in computer based instruction*. Doctoral dissertation, Florida State University.

Li, L., Liu, X., & Steckelberg, A. L. (2010). Assessor or assessee: How student learning improves by giving and receiving peer feedback. *British Journal of Educational Technology*, *41*(3), 525–536. https://doi.org/10.1111/j.1467-8535.2009.00968.x

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114* (pp. 9–18). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2459936.2459938

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education - ITICSE ’06* (Vol. 38, pp. 118–122). New York, New York, USA: ACM Press. https://doi.org/10.1145/1140124.1140157

Liu, N.-F., & Carless, D. (2006). Peer feedback: the learning element of peer assessment. *Teaching in Higher Education*, *11*(3), 279–290. https://doi.org/10.1080/13562510600680582

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 101–112). New York, NY, USA: ACM. https://doi.org/10.1145/1404520.1404531

Lu, J., & Law, N. (2012). Online peer assessment: Effects of cognitive and affective feedback. *Instructional Science*, *40*(2), 257–275. https://doi.org/10.1007/s11251-011-9177-2

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., … Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125. https://doi.org/10.1145/572139.572181

McKenney, S., & Reeves, T. C. (2014). Educational design research. In D. Jonassen, M. J. Spector, M. Driscoll, M. D. Merrill, J. van Merrienboer, & M. P. Driscoll (Eds.), *Handbook of Research on Educational Communications and Technology* (pp. 131–140). New York, NY: Springer.

Merrill, D. C., Reiser, B. J., Ranney, M., & Trafton, J. G. (1992). Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *The Journal of the Learning Sciences*, *2*(3), 277–305. https://doi.org/10.1207/s15327809jls0203_2

Miller, C. S. (2014). Metonymy and reference-point errors in novice programming. *Computer Science Education*, *24*(2–3), 123–152. https://doi.org/10.1080/08993408.2014.952500

Moos, D. C. (2011). Self-regulated learning and externally generated feedback with hypermedia. *Journal of Educational Computing Research*, *44*(3), 265–297. https://doi.org/10.2190/EC.44.3.b

Mory, E. (2004). Feedback research revisited. In D. Jonassen (Ed.), *Handbook of Research on Educational Communications and Technology Vol 2* (pp. 745–783). Mahwah, NJ: Lawrence Erlbaum Associates Publishers.

Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the First International Workshop on Computing Education Research* (pp. 57–67). New York, NY, USA: ACM. https://doi.org/10.1145/1089786.1089792

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 151–155). New York, NY, USA: ACM. https://doi.org/10.1145/1268784.1268830

Nelson, M. M., & Schunn, C. D. (2009). The nature of feedback: How different types of peer feedback affect writing performance. *Instructional Science*, *37*(4), 375–401. https://doi.org/10.1007/s11251-008-9053-x

Nicol, D. J., & Macfarlane-Dick, D. (2006). Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in Higher Education*, *31*(2), 199–218. https://doi.org/10.1080/03075070600572090

Özdemir, G., & Clark, D. B. (2007). An overview of conceptual change theories. *Eurasia Journal of Mathematics Science Technology Education*, *3*(4), 351–361. https://doi.org/10.12973/ejmste/75414

Patchan, M. M., & Schunn, C. D. (2015). Understanding the benefits of providing peer feedback: how students respond to peers' texts of varying quality. *Instructional Science*, *43*(5), 591–614. https://doi.org/10.1007/s11251-015-9353-x

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Pettit, R. S., Homer, J., & Gee, R. (2017). Do enhanced compiler error messages help students? *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17*, 465–470. https://doi.org/10.1145/3017680.3017768

Porter, L., Lee, C. B., & Simon, B. (2013). Halving fail rates using peer instruction: a study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 177–182). New York, NY, USA: ACM. https://doi.org/10.1145/2445196.2445250

Posner, G. J., Strike, K. A., Hewson, P. W., & Gertzog, W. A. (1982). Accommodation of a scientific conception: Toward a theory of conceptual change. *Science Education*, *66*(2), 211–227. https://doi.org/10.1002/sce.3730660207

Price, T. W., & Barnes, T. (2015). Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 91–99). New York, NY, USA: ACM. https://doi.org/10.1145/2787622.2787712

Price, T. W., Dong, Y., & Lipovac, D. (2017). iSnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17* (pp. 483–488). New York, NY, USA: ACM Press. https://doi.org/10.1145/3017680.3017762

Qian, Y., Hambrusch, S., Yadav, A., & Gretter, S. (2018). Who needs what: Recommendations for designing effective online professional development for computer science teachers. *Journal of Research on Technology in Education*, *50*(2), 164–181. https://doi.org/10.1080/15391523.2018.1433565

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*, *18*(1), 1:1-1:24. https://doi.org/10.1145/3077618

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*(3), 203–221. https://doi.org/10.1080/08993400500224310

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: A self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education*, *27*(1), 37–64. https://doi.org/10.1007/s40593-015-0070-z

Robins, A., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (pp. 165–173). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1151869.1151891

Sadler, D. R. (1989). Formative assessment and the design of instructional systems. *Instructional Science*, *18*(2), 119–144. https://doi.org/10.1007/BF00117714

Saeli, M., Perrenet, J., Jochems, W. M. G., & Zwaneveld, B. (2011). Teaching programming in secondary school: A pedagogical content knowledge perspective. *Informatics in Education*, *10*(1), 73–88. Retrieved from https://search.proquest.com/docview/864687645?accountid=13360

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, *15*(1), 59–82. https://doi.org/10.1080/08993400500056563

Sajaniemi, J., & Prieto, R. N. (2005). Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group* (pp. 145–159). Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Roles+of+Variables+in+Experts+'+Programming+Knowledge#0

Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented CS1 programs: concepts and misconceptions. *ACM SIGCSE Bulletin*, *39*(3), 166–170. https://doi.org/10.1145/1269900.1268834

Schroth, M. L. (1992). The effects of delay of feedback on a delayed concept formation transfer task. *Contemporary Educational Psychology*, *17*(1), 78–82. https://doi.org/10.1016/0361-476X(92)90048-4

Shulman, L. (1986). Those who understand: knowdge growth in teaching. *Educational Researcher*, *15*(2), 4–14.

Shute, V. J. (2008). Focus on formative feedback. *Review of Educational Research*, *78*(1), 153–189. https://doi.org/10.3102/0034654307313795

Simon. (2011). Assignment and sequence: why some students can't recognise a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (pp. 10–15). New York, NY, USA: ACM. https://doi.org/10.1145/2094131.2094134

Simon, B., Kohanfars, M., Lee, J., Tamayo, K., & Cutts, Q. (2010). Experience report: peer instruction in introductory computing. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 341–345). New York, NY, USA: ACM. https://doi.org/10.1145/1734263.1734381

Sirkia, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *12th Koli Calling International Conference on Computing Education Research* (pp. 19–28). https://doi.org/10.1145/2401796.2401799

Sleeman, D., Kelly, A. E., Martinak, R., Ward, R. D., & Moore, J. L. (1989). Studies of diagnosis and remediation with high school algebra students. *Cognitive Science*, *13*(4), 551–568. https://doi.org/10.1016/0364-0213(89)90023-2

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, *2*(1), 5–23. https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77

Smith, J. P., diSessa, A. A., & Roschelle, J. (1994). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, *3*(2), 115–163. https://doi.org/10.1207/s15327809jls0302_1

Smith, S., & Sherwood, B. (1976). Educational uses of the PLATO computer system. *Science*, *192*(4237), 344–352. https://doi.org/10.1126/science.769165

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. https://doi.org/10.1145/6592.6594

Sorva, J. (2012). *Visual program simulation in introductory programming education*. Doctoral dissertation, Aalto University, Espoo, Finland.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, *13*(4), 1–64. https://doi.org/10.1145/2490822

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632. https://doi.org/10.1145/6138.6145

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257–285. https://doi.org/10.1016/0364-0213(88)90023-7

Taber, K. S. (2013). *Modeling learners and learning in science education*. New York: Springer.

Taber, K. S. (2014). Alternative Conceptions/Frameworks/Misconceptions. In R. Gunstone (Ed.), *Encyclopedia of Science Education* (pp. 1–5). Dordrecht: Springer Netherlands. https://doi.org/10.1007/978-94-007-6165-0_88-2

Taylor, C., Zingaro, D., Porter, L., Webb, K. C., Lee, C. B., & Clancy, M. (2014). Computer science concept inventories: past and future. *Computer Science Education*, *24*(4), 253–276. https://doi.org/10.1080/08993408.2014.970779

Teague, D., & Lister, R. (2014). Programming: reading, writing and reversing. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education* (pp. 285–290). New York, NY, USA: ACM. https://doi.org/10.1145/2591708.2591712

Tew, A. E. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner*. Doctoral dissertation, Georgia Institute of Technology, Atlanta, GA, USA.

Ulloa, M. (1980). Teaching and learning computer programming: A survey of student problems, teaching methods, and automated instructional tools. *ACM SIGCSE Bulletin*, *12*(2), 48–64. https://doi.org/10.1145/989253.989263

Vainio, V., & Sajaniemi, J. (2007). Factors in novice programmers’ poor tracing skills. *ACM SIGCSE Bulletin*, *39*(3), 236–240. https://doi.org/10.1145/1269900.1268853

Van der Kleij, F. M., Feskens, R. C. W., & Eggen, T. J. H. M. (2015). Effects of feedback in a computer-based learning environment on students’ learning outcomes: A meta-analysis. *Review of Educational Research*, *85*(4), 475–511. https://doi.org/10.3102/0034654314564881

VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*. https://doi.org/10.1080/00461520.2011.611369

VanLehn, K., Graesser, A. C., Jackson, G. T., Jordan, P., Olney, A., & Rosé, C. P. (2007). When are tutorial dialogues more effective than reading? *Cognitive Science*, *31*(1), 3–62. https://doi.org/10.1080/03640210709336984

Vosniadou, S. (1994). Capturing and modeling the process of conceptual change. *Learning and Instruction*, *4*(1), 45–69. https://doi.org/10.1016/0959-4752(94)90018-3

Vosniadou, S. (2013). Conceptual change in learning and instruction: The framework theory approach. In S. Vosniadou (Ed.), *International Handbook of Research on Conceptual Change* (pp. 11–30). New York, NY: Taylor and Francis.

Vosniadou, S., & Skopeliti, I. (2014). Conceptual change from the framework theory side of the fence. *Science & Education*, *23*(7), 1427–1445. https://doi.org/10.1007/s11191-013-9640-3

Webb, M., Davis, N., Bell, T., Katz, Y., Reynolds, N., Chambers, D. P., & Sysło, M. M. (2017). Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education and Information Technologies*, *22*(2), 445–468. https://doi.org/10.1007/s10639-016-9493-x

Weintrop, D., & Wilensky, U. (2015). Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 101–110). New York, NY, USA: ACM. https://doi.org/10.1145/2787622.2787721

Wentling, T. L. (1973). Mastery versus nonmastery instruction with varying test item feedback treatments. *Journal of Educational Psychology*, *65*(1), 50–58. https://doi.org/10.1037/h0034820

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (pp. 243–252). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1151869.1151901

Xu, S., & Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, *29*(4), 360–384. https://doi.org/10.1109/TSE.2003.1191799

Yadav, A., Berges, M., Sands, P., & Good, J. (2016). Measuring computer science pedagogical content knowledge: An exploratory analysis of teaching vignettes to measure teacher knowledge. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education,* (October), 92–95. https://doi.org/10.1145/2978249.2978264

Yang, M., Badger, R., & Yu, Z. (2006). A comparative study of peer and teacher feedback in a Chinese EFL writing class. *Journal of Second Language Writing*, *15*(3), 179–200. https://doi.org/10.1016/j.jslw.2006.09.004

APPENDIX A. COURSE SYLLABUS

Programming and Computational Thinking

Summer Residential Syllabus
Teacher: Yizhou Qian

Goal:

This course will help students learn the fundamental syntax of Java programming. Meanwhile, this course focuses on the development of students' computational thinking, which is the skill of the 21st century. After receiving this course, students at least will able to design programs to solve simple problems, such as some math problems which are difficult to solve by hand. In addition, the course is a good preparation of AP CS A course.

Teaching Methods:

- Project-based

Students will learn Java through completing an individual project. Every student needs to design a game and after finishing this game, they could master most of the syntax of programming. Then they will start a team project according to their interest and programming ability.

- Learning scaffolding by technology-based learning environment

A learning support system to help students practice their programming skill will be used. This system is game-based designed. When they complete the learning tasks in the system, students not only improve their programming skill, but also the computational thinking.

Week 1

Day / Date	**Topic**
Day 1	➢ Introduction about the course ➢ Build your first program
Day 2	➢ Introduction to Java Programming ➢ Variables ➢ Operators & Operations ➢ Free Practice
Day 3	➢ Review the learned content ➢ Conditionals (If Statements) ➢ Free Practice
Day 4	➢ Review the learned content ➢ Loops (While Loop) ➢ Free Practice
Day 5	➢ Review the learned content ➢ Loops (For Loop) ➢ Free Practice

Week 2

Day	Topic
Day 6	➢ Review the learned content ➢ Introduce and Work on the Individual Project
Day 7	➢ Work on the Individual Project ➢ Presentations
Day 8	➢ Introduce and Work on the Secret Message Project ➢ Free Practice
Day 9	➢ Introduce and Work on Final Project - Gladiator ➢ Test Final Projects
Day 10	➢ Revise Final Projects ➢ Gladiator Fights ➢ Course Review

APPENDIX B. FEEDBACK MESSAGES

Feedback for Common Compilation Errors

#	Feedback Message	Relevant Error
CFB1	You may have **mismatched or missing** braces **{}**, quotation marks **""**, parens **()**, or brackets **[]** in your code. Make sure you have them in pairs.	CE4: class expected CE5: reached end of file CE7:) expected CE9: identifier expected
CFB2	You may have **typos**, **code in wrong place**, or **incomplete code** in your program. Make sure you use and spell variables and statements correctly.	CE1: cannot find symbol CE6: not a statement CE8: illegal start of expression CE13: illegal start of type
CFB3	You may miss **semicolon ;** somewhere in your code. Check if you use **semicolon ;** appropriately.	CE2: ; expected
CFB4	The name of your program is wrong!	CE3: program name error

CFB5	**The type of a variable has to match its value.** Your program may have mismatched type and value of variables. The following code provides an example of this **error**:
	`// Try to assign String to int`
	`int a = in.nextLine();`
	`// Try to assign int to String`
	`String b = in.nextInt();`
		CE10: incompatible types

CFB6	**A variable can only be defined once.** Your program may define a variable twice. The following code provides an example of this **error**:
	`// Define variables`
	`int a = 10;`
	`int b = 20;`
	`// Try to define the variable a again`
	`int a = b + 30;`
		CE11: variable is already defined

CFB7	You may use operator(s) in a wrong way!	CE12: incorrect use of operators

CFB8	You may try to assign a **double** value to an **int** variable. This leads to a possible loss of precision. The following code provides an example of this **error**:
	`double pi = 3.14;`
	`// assign double to int`
	`int b = 2 * 2 * pi;`
		CE15: possible loss of precision

Feedback for Common Test Errors

#	Feedback Message	Relevant Error

TFB1	The user may enter a number such as 2.3. Your program has to read a **double** instead of an **int**.
	The following code may help you solve your problem:
	`Scanner in = new Scanner(System.in);`
	`double radius = in.nextDouble();`
		TE1: Mismatched input (Problem: Area of Circle)

TFB2	You may forget to use **String.format("%.2f", area)** to display only **2 decimal places** of a double. **Or you print the wrong variable**.
	Here is the example code to solve this issue:
	`String result = String.format("%.2f", area);`
	`System.out.println( result );`
		TE2: Wrong decimal places (Problem: Area of Circle)

TFB3	There is **a space** after the **comma ,** There is an **exclamation mark !** at the end of the output.	TE3 Missing punctuation (Problem: Say Hi to Anyone)
TFB4	An integer divided by another integer gives you an integer in Java. For example, 11 / 2 gives 5. However, 11 / **2.0** gives you 5.5 The following code may help you solve your problem: `double s = (a + b + c) / 2.0;`	TE4: Integer division issue (Problem: Area of Triangle)

TFB5	You may forget to use **String.format("%.2f", area)** to display only **2 decimal places** of a double. **Or you print the wrong variable**.
	Here is the example code to solve this issue:
	`String result = String.format("%.2f", area);`
	`System.out.println( result );`
		TE5: Wrong decimal places (Problem: Area of Triangle)

TFB6	Please try the input cases such as **1 4 7** and **1 4 4** to check the output of your program. You may want to consider the problem in this way - **(b*b) - 4*a*c** is called **Discriminant** • When Discriminant is **positive**, there will be two solutions. • When Discriminant is **zero**, there will be only one solution. • When Discriminant is **negative**, there will be no answer. Note: you should not use Math.sqrt() on a negative number, e.g., Math.sqrt(-3.14).	TE6: Inappropriate comparison (Problem: Quadratic Equation 2)
TFB7	If there are two different roots, print the smaller one the first line and the larger one on the second line.	TE7: Wrong output (Problem: Quadratic Equation 2)
TFB8	Note: The user will only enter **one integer** with three digits (e.g., 100, 911). Try to use operators such as / **(Division)** and **%**	TE8: Mismatched input (Problem: Sum of Digits)

	(Modulus) to get each digit of the integer. For example: 234 / 100 will give 2 234 % 100 will give 34 234 % 10 will give 4	

TFB9	In this problem, the user will enter **two integers**. Your program failed to read two integers from the user.
	The following code is an example of reading two integers:
	`Scanner in = new Scanner(System.in);`
	`int a = in.nextInt();`
	`int b = in.nextInt();`
		TE9: Mismatched input (Problem: How Old Are We?)

TFB10	Your program may not have output in some cases. Try the input cases **2 3 3** and **7 7 7** and fix the problems of your program.	TE10: Forgot Special Cases (Problem: Who is Max?)