Using Targeted Feedback to Address Common Student Misconceptions in Introductory Programming: A Data-Driven Approach

Yizhou Qian¹ and James D. Lehman²

Abstract
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. Digital tools, such as automated assessment systems, can be useful and supportive in teaching CS courses. This two-stage design-based research (DBR) study investigated the effects of targeted feedback in an automated assessment system for addressing common misconceptions of high school students in a Java-based introductory programming course. 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 and facilitate learning. Although the findings of this exploratory study showed evidence of the power of digital tools, more research is needed to make technology benefit more CS teachers.

Keywords
introductory programming, misconceptions, targeted feedback, computer science education, pedagogical content knowledge (PCK)

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 CS courses in their K-12 curriculum, such as the United States, the United Kingdom, and others (Brown, Sentance, Crick, & Humphreys, 2014; desJardins, 2015). Introductory CS courses, however, are difficult for beginners (Guzdial, 2015; McCracken et al., 2001), and students often exhibit misconceptions that impede their learning of programming (Altadmiri & Brown, 2015; Qian & Lehman, 2017; Sorva, 2013). CS teachers in K-12 schools should be prepared to help students establish accurate understanding of CS and programming (Qian & Lehman, 2017). One means of supporting CS teachers’ instruction is to use technology such as an automated assessment system, which is a tool that can automatically evaluate the correctness of students’ programs and provide immediate feedback (Doucet, Livingstone, & Orwell, 2005; Gerdes, Heeren, Jeuring, & van Binsbergen, 2017). This exploratory design-based research study investigated the effects of targeted feedback in an automated assessment system for addressing common misconceptions of high school students in a Java-based introductory programming course.

Literature Review
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). Previous studies on student misconceptions in introductory programming have cataloged a broad range of
student misconceptions including syntax errors and other difficulties caused by misconceptions (Altadmri & Brown, 2015; Denny, Luxton-Reilly, & Tempero, 2012; du Boulay, 1986; Guzdial, 1995; Kaczmarsczyk, Petrick, East, & Herman, 2010; Qian & Lehman, 2017; Ragonis & Ben-Ari, 2005; Simon, 2011; Sleeman, Putnam, Baxter, & Kuspa, 1986; Sorva, 2013). For instance, novice students often make syntactic mistakes in their code, such as mismatching parentheses, missing semicolons, failing to declare variables, 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). 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 (Kaczmarszyk et al., 2010; Sleeman et al., 1986). In addition, students usually lack well-established programming strategies (Clancy & Linn, 1999; Davies, 1993; Lister, Simon, Thompson, Whalley, & Prasad, 2006; Sajaniemi & Prieto, 2005; Soloway, 1986) leading to difficulties with planning, composing, and debugging programs.

To address students’ misconceptions in introductory programming, researchers and educators have developed various instructional tools, 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, which have 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, Pardo, & Delgado Kloos, 2013; Gerdes et al., 2017). Using 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.

Although such systems seem to be an ideal solution to help teachers identify and address student misconceptions, they are difficult to develop, 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 seeing only the raw Java compiler error messages. Although these results are promising, there are two issues with prior studies on automated assessment systems with such feedback components. 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).

Misconceptions and Conceptual Change Theories

In science and mathematics education, researchers and educators have developed conceptual change theories to understand the development of student conceptions. 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).

Two 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 naive 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). From this viewpoint, 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). Thus, 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, 2014).

Although debate between the two perspectives is ongoing (see diSessa, 2013; Vosniadou, 2013), the current trend in conceptual change research has shown convergence (Vosniadou & Skopeliti, 2014). Nowadays, researchers of conceptual change theories share the ideas that (a) learners’ own preinstructional 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). Before students successfully understand the new academic concept, the interaction between the new concept and their existing knowledge results in various synthetic models, which are intermediate states of knowledge with partially correct interpretation (Vosniadou, 1994; Vosniadou & Skopeliti, 2014). Hence, success in conceptual change requires tracking the development of learners’ (mis)conceptions using real-time data (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 (Taber, 2014).

Although 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). This study applied conceptual change theories to understand student misconceptions in introductory programming.

Feedback for Conceptual Change

Feedback is information provided by an agent to correct learners’ errors and misunderstandings for the purpose of facilitating learning (Butler & Winne, 1995; Hattie & Timperley, 2007; Kulhavy & Wager, 1993; Shute, 2008). Researchers have developed different models of feedback to explain how feedback facilitates learning and provide guidelines for designing effective feedback (Bangert-Drowns, Kulik, Kulik, & Morgan, 1991; Clariana, Wagner, & Murphy, 2000; Hattie & Gan, 2011; Kulhavy & Stock, 1989). One widely accepted model is the visibility model of feedback focusing on visualizing learners’ current knowledge states (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, pp. 257-258). Visibility means that effective feedback design needs to make the discrepancy visible to both the instructor and the learner. According to Hattie and Gan (2011), the problem with traditional feedback design is that it neglects to examine learners’ current (mis)conceptions. Effective feedback design requires scrutinizing learners’ erroneous responses, to grasp their common misconceptions, and provide corrective information targeted at addressing misconceptions (Hattie & Gan, 2011; Hattie & Timperley, 2007). Procedures for designing effective feedback include (a) clearly describing the desired learning outcomes, (b) precisely analyzing learners’ current knowledge states, and (c) identifying students’ misconceptions and providing information for promoting conceptual change and enhancing learning (Hattie & Gan, 2011; Hattie & Timperley, 2007).

Summary

Previous studies have cataloged a broad range of student misconceptions and explored the effects of a variety of tools that can help to address student misconceptions. However, most of the studies have focused on college students (e.g., Altadmri & Brown, 2015; Becker, 2016; Jackson et al., 2005; Pettit et al., 2017). This study focused on misconceptions among high school students taking an introductory programming course. Moreover, although many automated assessment systems have been developed and tested by researchers, most systems to date either provide direct feedback for correcting simple errors in code (e.g., Gerdes et al., 2017) or provide feedback based only on compiler errors (e.g., Becker, 2016). Furthermore, the effectiveness of using enhanced compiler error messages as feedback is still questionable (Denny et al., 2014; Pettit et al., 2017). This study focused on designing and providing targeted feedback for addressing student misconceptions based on the analysis of both compilation and test errors in students’ programs. Finally, this study adopted conceptual change theories and the visibility model of feedback as the theoretical framework for understanding and addressing student misconceptions. Conceptual change theories suggest using learner data to understand current states of student (mis)conceptions before choosing proper strategies for accomplishing conceptual change. According to the visibility model, the first step to design effective feedback for promoting conceptual change is to analyze the discrepancy between the students’ current knowledge states and the intended outcomes using learner data. Both conceptual change theories and the visibility model emphasize the importance of understanding current states of learner knowledge and tracking the evolution of student (mis)conceptions using learner data. Although previous studies in CS education have discussed student misconceptions from a variety of perspectives, little work has drawn on our understanding of conceptual change and appropriate use of data-driven feedback to promote conceptual change (Qian & Lehman, 2017).

Purpose of the Study

The purpose of this study was to investigate how targeted feedback based on student data in an automated assessment
system affected the evolution of common (mis)conceptions of high school students in an introductory programming course. The following research questions guided the study:

**Research Question 1 (RQ1):** Does targeted feedback have positive effects on addressing common student misconceptions?

**Research Question 2 (RQ2):** How does targeted feedback for promoting conceptual change affect students’ (mis)conceptions?

**Method**

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; McKenney & Reeves, 2014). DBR studies seek to design and test interventions iteratively in classroom settings using both quantitative and qualitative data and develop principles or theories for helping others facing similar situations (Anderson & Shattuck, 2012; McKenney & Reeves, 2014). This exploratory DBR study consisted of two stages and investigated the effects of targeted feedback in an automated assessment system for addressing common misconceptions of high school students in a Java-based introductory programming course.

**Participants and Research Settings**

Participants in this study were two groups of high school students, a total of 23, who took a Java-based introductory programming class in two different sections of a summer residential program. This summer residential program has been offered by an established center (GERI) for gifted and talented students at a major university in the U.S. Midwest. The 2017 summer program consisted of two 2-week sections. Section 1 was from July 2 to July 15, 2017. Section 2 was from July 16 to July 29, 2017. 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 (GERI Website, 2018). Group 1 of this study, students who were in Section 1, originally had 15 students, and Group 2, students who were in Section 2, had 10 students. However, one student of Group 1 was found to have cheated when solving problems and another student of Group 1 was an outlier in terms of ability who was the champion of a programming competition in his hometown and solved all the problems in the automated assessment system within 2 days. 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 in Group 1 and 10 students in Group 2. Although the number of participants in this exploratory study was not large, as explained in section “Results,” the participants created a substantial data set of student problem solutions for analysis.

The automated assessment system used in this study was called Mulberry, which is designed for Java learners and developed by the author. Compared with existing automated assessment systems, Mulberry has two distinct features. First, Mulberry is designed and developed using gamification principles. Students need to create avatars to solve problems in Mulberry. When solving a problem, a student’s avatar gains experience points and rewards (represented as gold). When the student’s avatar accumulates sufficient experience points, his or her avatar will level up and unlock more difficult tasks. Figure 1 shows the major user interface (UI) of Mulberry. This gamification design can increase students’ motivation of using the system to practice programming. Second, Mulberry provides targeted feedback when students make mistakes in their code. Most existing automated assessment systems only provide assessment results to inform students about the correctness of their programs (e.g., failed to compile). Some systems such as Decaf provide elaborated feedback for common compilation errors to help students fix their programs. The distinct feature of Mulberry is that it provides elaborated feedback messages for both common compilation and noncompilation errors in students’ code. This is an advance over other similar systems, and the effect of the feedback component is the focus of this study.

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. Every problem has several test cases, which are pairs of input data and expected output. Figure 2 shows an example problem in Mulberry and its test cases. Mulberry automatically assesses students’ solutions to each problem 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.

The Java-based introductory programming class in this study was called Programming and Computational Thinking and was offered to high school students in both sections of the summer residential program in the morning from 8:30 to 11:30 every weekday. The researcher was the instructor of the course. The major topics covered in this course were Program Structure, Input/Output (I/O), Variables and Operators, Conditionals, and Loops. Typically, during every class session, the instructor started with a 30-min lecture to review previously learned content and introduce new course content. 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 Mulberry. After all the topics were introduced (the first week), students started individual and team projects based on their choices (e.g., design a text-based interactive game). 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.

**Procedures**

In the first stage, students of group 1 took the introductory programming class and used Mulberry to practice their
programming skills. The goal of the first stage was to identify common misconceptions students exhibited in the introductory programming course. When students of Group 1 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 students’ common misconceptions in this introductory programming course using all the student solutions collected by Mulberry, including student solutions of Group 1 and previous student solutions in Mulberry. As the focus of this article is the effects of targeted feedback for addressing common student misconceptions in this introductory programming course, details of how the common misconceptions were identified are not reported here. See Qian (2018) for detailed procedures and results.

Table 1 presents the identified common errors and relevant misconceptions.

In the second stage, students of Group 2 took the same introductory programming class. When they solved problems in Mulberry and submitted solutions producing incorrect output, they received the targeted feedback to address their misconceptions. Before students of Group 2 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; Vosniadou & Skopeliti, 2014) 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.

Table 1. Common Errors and Relevant Misconceptions (Qian, 2018).

| Common compilation errors and relevant misconceptions |
|------------------------------------------------------|
| # | Compilation error | Relevant misconceptions |
|---|--------------------|-------------------------|
| CE1 | cannot find symbol | Confusion about Java variables and variable operations |
| CE2 | ; expected | Deficient knowledge of fundamental program structure |
| CE3 | program name error | |
| CE4 | class expected | |
| CE5 | reached end of file | |
| CE6 | not a statement | |
| CE7 | ) expected | |
| CE8 | illegal start of expression | Misunderstandings of Java expressions |
| CE9 | identifier expected | |
| CE10 | incompatible types | |
| CE11 | variable is already defined | |
| CE12 | incorrect use of operators | |
| CE13 | illegal start of type | |
| CE14 | { expected | |
| CE15 | possible loss of precision | |

| Common test errors and relevant misconceptions |
|------------------------------------------------|
| Difficult problem | # | Test error | Relevant misconceptions |
|-------------------|---|------------|-------------------------|
| Area of Circle | TE1 | Mismatched input | Misunderstandings of Java input |
| | TE2 | Wrong decimal places | |
| Say Hi to Anyone | TE3 | Missing punctuation | Misunderstandings of Java output |
| Area of Triangle | TE4 | Integer division issue | Confusion about Java operators |
| | TE5 | Wrong decimal places | Misunderstandings of Java output |
| Quadratic Equation 2 | TE6 | Inappropriate comparison | Confusion about Java operators |
| | TE7 | Wrong output | Misunderstandings of Java output |
| Sum of Digits | TE8 | Mismatched input | Misunderstandings of Java input |
| How Old Are We? | TE9 | Mismatched input | Misunderstandings of Java input |
| Who is Max? | TE10 | Forgot Special Cases | Forgetting to consider special cases |
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 \texttt{; expected} error was typically caused by missing a required semicolon(s), and students received the following targeted feedback message in Mulberry:

> 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 \texttt{reached end of file} and \texttt{) 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.

> The common compilation error \texttt{)} expected seems to be a similar error, but its cause may be very complicated and often is irrelevant to a missing opening brace \texttt{)}. To not mislead students, no targeted feedback was designed and provided for the \texttt{)} 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, a general feedback message was provided (see CFB2 in the appendix). In the end, eight unique feedback messages were designed for 14 common compilation errors (see the appendix).

**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, the \texttt{Area of Triangle} problem was a difficult problem, and 48% of students who solved it showed the same test error that produced the wrong output \texttt{3.16} when the input was \texttt{3 4 4}. Although students might have written very different incorrect solutions, their key errors were identical. Figure 3 illustrates two solutions that were written by two different students but produced the same wrong output \texttt{3.16}. In these cases, both students failed to recognize that the result of the expression \texttt{“(a + b + c) / 2”} would be an integer without decimal values. The misconception in this example is that students had a problematic understanding of the Java division operator. Thus, when a student solution to the \texttt{Area of Triangle} problem had the common test error illustrated in Figure 3, he or she received the following feedback message:

> 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. In the end, 10 unique feedback messages were designed for the common test errors (see the appendix).

**Data Analysis**

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 addressing common student misconceptions to answer RQ1. To check the effects of feedback, erroneous student solutions of both Group 1 and Group 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 labeled 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 labeled as improved. When an erroneous solution had compilation errors, and its next solution also had compilation errors, it was labeled not improved. When an erroneous solution had test errors, and its next solution had compilation or test errors, it was also labeled 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 to answer RQ2. 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, a feedback improvement rate (FIR) was calculated using the following formula:

\[ FIR = \frac{\text{the number of improved solutions with the feedback}}{\text{the number of occurrences of the feedback}} \]

Feedback messages with the best and worst FIR 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 so had FIRs of 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 Group 2 were extracted from the Mulberry database. Although 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 4 shows an example of how a student revised his or her solutions to the \textbf{Area of Triangle} problem. In Solution #1, the student encountered a syntax error, because the return value of the \texttt{Math.sqrt()} method is a double rather than an integer. In

\begin{figure}
\centering
\includegraphics[width=\textwidth]{solution1.png}
\caption{Solution #1}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{solution2.png}
\caption{Solution #2}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{solution3.png}
\caption{Solution #3}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{solution4.png}
\caption{Solution #4}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{solution5.png}
\caption{Solution #5}
\end{figure}
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.

Results

**RQ1: Does Targeted Feedback Have Positive Effects on Addressing Common Student Misconceptions?**

**Difference in overall improvement rates.** Students’ solutions of the two groups were used to analyze the overall 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 5). 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, \(\chi^2(1, N = 918) = 11.11, p < .001\). 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 6). The results of a chi-square test indicated that the difference was significant, $\chi^2(1, N = 480) = 17.45, p < .001$. 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.

**RQ2: How Does Targeted Feedback for Promoting Conceptual Change Affect the Evolution of Students’ (Mis)conceptions?**

Qualitative analysis of student code was used to address RQ2. Four feedback cases were selected for the qualitative analysis according to the case selection procedures described in the “Method” section. First, for each feedback message, its FIR was calculated. Table 2 presents the FIRs of feedback messages for compilation and test errors. In addition to the FIR, the number of occurrences of the errors and the number of improvements are also included in the tables (see the numbers within the parentheses). Although students of Group 1 did not receive targeted feedback messages for the common errors, their FIRs are also presented in the tables 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 3. According to the case selection procedures, the four selected cases were CFB4, CFB3, TFB4, and TFB1. CFB4 and CFB3 were the compilation error feedback messages (CE Feedback) with the best and worst FIR. TFB4 and TFB1 were the test error feedback messages (TE Feedback) with the best and worst FIR. Table 3 presents the details about the four feedback messages.

**Compilation error feedback message with best FIR.** The feedback message for addressing the program name error showed an FIR 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 3).

The analysis of relevant student solutions found different patterns of improving the code between students of Group 2 and Group 1. 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 8 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 more intermediate solutions to fix this error. For instance, Tim, a student of Group 1, named his program as `Project01` while the required program name was `HelloAnyone`. In the next solution, he deleted the whole class definition line, rather than revising the name. As he only received the default compiler message, he might see an error message like “Error: class Project01 is public, should be declared in a file named Project01.java.” With this error message, novices might not be able to understand what exactly was wrong. In this case, Tim might mistakenly believe that the line `public class Project01` was wrong and should be deleted. Although he eventually fixed this error, he submitted wrong intermediate solutions.

Although 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 FIR.** The feedback message for addressing the `; expected` error was identified as the worst case. The FIR 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.

Figure 9 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. Although 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. Although 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 FIR. The feedback message with the best FIR 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.

According to the quantitative data, students of Group 2 made this error 6 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. 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.

When students in Group 1 tried to fix this error, they tended to have more intermediate solutions. For example, one student Emily made this error in her program. 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 FIR. The feedback message with the worst FIR 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.

According to the quantitative data, students of Group 2 made this error 6 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 actually fixed this error but still had other errors. Figure 10 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. This caused Solution #2 to 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 only three successful improvements. Among the other 25 failed cases, only three identified the error immediately and made some partial improvements. Most students made this error in one solution, but they did not fix this error in the next solution and made several erroneous intermediate solutions. If the feedback message had been presented, they might not have required those intermediate solutions.

**Discussion and Conclusion**

**Overall Effects of Targeted Feedback**

The results of this study indicated that targeted feedback messages enhanced students’ improvement rates of erroneous solutions to programming problems. 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 with 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 group receiving feedback messages made 32% fewer errors than the group only seeing the original Java compiler error messages. Although the study of Becker et al. (2016) only investigated students’ compilation errors, its overall research approach and results are similar to this study. However, two prior studies reported ineffectiveness of feedback using enhanced compiler error messages (Denny et al., 2014; Pettit et al., 2017), but their research design was different from Becker et al.’s (2016) and this study, and this 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). In the study of Pettit et al. (2017), their feedback messages only covered 30% of compilation errors, which may make the effects of their feedback insignificant. Although 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.

In this study, targeted feedback messages were designed and provided for both common compilation and test errors. The results suggest that one important step of designing the feedback component of an automated assessment system is to identify the common errors students make, which are representative of common difficulties students encounter in learning to programming. Without a good identification of common student errors that 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 showed 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. In contrast, with the targeted feedback message, students of Group 2 often could completely fix the same error in the revised solution (see Figure 10 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 Group 2 formed certain melded concepts, but the targeted feedback messages appear to 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’ 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 may have worked even 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” because some errors were still present even though the error related to the feedback was fixed. 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 (PCK; Hubwieser, Magenheim, Mühlung, & Ruf, 2013; Shulman, 1986; Yadav, Berges, Sands, & Good, 2016). PCK refers to the knowledge that enables teachers to transform instructional content into a comprehensible form to students (Shulman, 1986). Teachers’ knowledge of and ability to address student misconceptions is a key component of their PCK (Carlson, 1999; Saefli, Perrenet, Jochems, & Zwaneveld, 2011; Shulman, 1986). Unfortunately, research on CS teachers’ PCK is limited (Saefli et al., 2011; Yadav et al., 2016), and CS teachers often lack sufficient knowledge of student misconceptions (Brown & Altadmri, 2017; Guzdial, 2015). The results of this study indicate that designing feedback based on common student misconceptions by analyzing student data in an automated assessment system can be effective to address student misconceptions and has the potential to help teachers develop their PCK.

Automated assessment systems have been widely used in 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 common student misconceptions (Becker, 2016; Denny et al., 2012; Qian & Lehman, 2017). Based on the common misconceptions, designing and adding targeted feedback using the approach suggested by this study can be effective to address student misconceptions and facilitate learning. Hence, researchers and developers of automated assessment systems should consider the common misconception identification component and the targeted feedback design component as built-in components when designing and improving their systems. Meanwhile, future professional development programs for CS teachers should pay attention to teachers’ ability to use automated assessment systems and help them learn to design and provide targeted feedback based on common student errors and difficult problems using student data in such systems. We believe the ability to effectively identify and address common misconceptions based on student data will be vital to quality CS teachers.

**Limitations**

Although 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 in a non-school-based summer enrichment program that was not formally graded. Their misconceptions may not be representative of the population of high school students in formal educational settings.

In addition, 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.

Finally, the number of participants of this study was relatively small. We only analyzed 23 students’ programs to investigate the effects of targeted feedback. However, the overall size of the data set used in this research project was not small. The design of the targeted feedback messages was based on the common misconception analysis of 4,873 student solutions from 55 students (see Qian, 2018 for details). Moreover, we analyzed 928 student solutions to examine the effects of targeted feedback; in other words, about 40 solutions per student were analyzed. In addition, because our research involved a manually implemented qualitative component to assess how the feedback affected students’ conceptual change, a larger sample was not feasible. This exploratory study was a “proof-of-concept” study, in which we tested the targeted feedback design model. We believe that our overall data set size was substantial and sufficient for the purpose of the study. For our future research, we will test targeted feedback component with a larger number of participants.

**Conclusion**

With the expansion of CS education, CS teachers in K-12 schools should be cognizant of student misconceptions and be prepared to help students establish accurate understanding of CS and programming. Digital tools, such as automated assessment systems, definitely can be useful and supportive in teaching CS courses. This two-stage DBR study investigated the effects of targeted feedback in an automated assessment system for addressing common misconceptions of high school students in a Java-based introductory programming course. 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 and facilitate learning. Although the findings of this exploratory study showed evidence of the power of digital tools, more research is needed to make technology benefit more CS teachers.
## Appendix

### 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       |
|     | **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 |

Note. CE = compilation error.
### 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 % (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 | TE8: Mismatched input (Problem: Sum of Digits) |
| 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?) |

TE = test error.
Declaration of Conflicting Interests
The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding
The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Fundamental Research Funds for the Central Universities under grant JUSRP11982 and 2019JDZD08.

ORCID ID
Yizhou Qian https://orcid.org/0000-0003-2551-9581

References
Abimbola, I. O. (1988). The problem of terminology in the study of student conceptions in science. Science Education, 72, 175-184. doi:10.1002/sce.3730720206
Altadmri, A., & Brown, N. C. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In A. Decker & K. Eiselt (Eds.), Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 522-527). New York, NY: ACM Press. doi:10.1145/2676723.2677258
Anderson, T., & Shattuck, J. (2012). Design-based research: A decade of progress in education research? Educational Researcher, 41, 16-25. doi:10.3102/0013189X11428813
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, 213-238. doi:10.3102/00346543061002213
Barnes, T., & Stamper, J. (2010). Automatic hint generation for logic proof tutoring using historical data. Educational Technology and Society, 13, 3-12.
Becker, B. A. (2016). An effective approach to enhancing compiler error messages. In C. Alphonce & J. Tims (Eds.), Proceedings of the 47th ACM Technical Symposium on Computer Science Education—SIGCSE’16 (pp. 126-131). New York, NY: ACM Press. doi: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, 148-175. doi:10.1080/08993408.2016.1225464
Brown, N. C. C., & Altadmri, A. (2017). Novice Java programming mistakes: Large-scale data vs. educator beliefs. ACM Transactions on Computing Education, 17(2), 71. doi: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, 1-22. doi:10.1145/2602484
Butler, D. L., & Winne, P. H. (1995). Feedback and self-regulated learning: A theoretical synthesis. Review of Educational Research, 65, 245-281. doi: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). New York, NY: Kluwer Academic.
Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. ACM SIGCSE Bulletin, 31, 37-42. doi:10.1145/384266.299673
Clarina, R. B., Wagner, D., & Murphy, L. C. R. (2000). Applying a connectionist description of feedback timing. Educational Technology Research and Development, 48, 5-22. doi:10.1007/BF02319855
Davies, S. P. (1993). Models and theories of programming strategy. International Journal of Man-Machine Studies, 39, 237-267. doi:10.1006/imms.1993.1061
De-La-Fuente-Valentin, 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, 33-42. doi:10.1016/j.compedu.2012.09.004
Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. In A. Cajander & M. Daniels (Eds.), Proceedings of the 19th ACM Conference on Innovation & Technology in Computer Science Education (pp. 273-278). New York, NY: ACM Press. doi:10.1145/2591708.2591748
Denny, P., Luxton-Reilly, A., & Tempero, E. (2012). All syntax errors are not equal. In T. Lapidot & J. Gal-Ezer (Eds.), Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education—ITiCSE’12 (pp. 75-80). New York, NY: ACM Press. doi:10.1145/2325296.2325318
desJardins, M. (2015). Creating AP® CS principles: Let many flowers bloom. ACM Inroads, 6, 60-66. doi:10.1145/2835852
diSessa, A. A. (1993). Toward an epistemology of physics. Cognition and Instruction, 10, 105-225. doi: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 & Francis.
diSessa, A. A. (2014). The construction of causal schemes: Learning mechanisms at the knowledge level. Cognitive Science, 38, 795-850. doi:10.1111/cogs.12131
Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming. Journal on Educational Resources in Computing, 5, 41-13. doi:10.1145/1163405.1163409
du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2, 57-73. doi: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, 671-688. doi:10.1080/09500690305016
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 C. Alphonce & J. Tims (Eds.), Proceedings of the 47th ACM Technical Symposium on Computing Science Education (pp. 150-155). New York, NY: ACM Press. doi:10.1145/2839509.2844631
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, 65-100. doi:10.1007/s40593-015-0080-x
Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for CS education. In T. Camp & P. Tymann (Eds.), SIGCSE 2013—Proceedings of the 44th ACM Technical Symposium on Computer Science Education (pp. 579-584). New York, NY: ACM Press. doi:10.1145/2445196.2445368

Guzdial, M. (1995). Centralized mindset: A student problem with object-oriented programming. ACM SIGCSE Bulletin, 27, 182-185. doi:10.1145/999691.999772

Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. Synthesis Lectures on Human-Centered Informatics, 8, 1-165. doi:10.2200/S00684ED1V01Y201511HCI003

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, 81-112. doi:10.3102/003465430298487

Hubwieser, P., Magenheim, J., Mühlung, A., & Ruf, A. (2013). Towards a conceptualization of pedagogical content knowledge for computer science. In B. Simon, A. Clear, & Q. Cutts (Eds.), Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research—ICER’13 (pp. 1-8). New York, NY: ACM Press. doi:10.1145/2493394.2493395

Jackson, J., Cobb, M., & Carver, C. (2005). Identifying top Java errors for novice programmers. In W. Oakes, D. Voltmer, & C. Yokomoto (Eds.), Proceedings Frontiers in Education 35th Annual Conference (p. T4C). New York, NY: IEEE. doi:10.1109/FIE.2005.1611967

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In G. Lewandowski & S. Wolfman (Eds.), Proceedings of the 41st ACM Technical Symposium on Computer Science Education (pp. 107-111). New York, NY: ACM Press. doi: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, 37, 83-137. doi:10.1145/1089733.1089734

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, 41-84. doi:10.1016/j.jvlc.2004.08.003

Kulhavy, R. W., & Stock, W. A. (1989). Feedback in written instruction: The place of response certitude. Educational Psychology Review, 1, 279-308. doi: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.

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 R. Davoli, M. Goldweber, & P. Salomoni (Eds.), Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education—ITICSE’06 (Vol. 38, pp. 118-122). New York, NY: ACM Press. doi:10.1145/1140124.1140157

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, 125-180. doi: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.

Özdemir, G., & Clark, D. B. (2007). An overview of conceptual change theories. Eurasia Journal of Mathematics Science Technology Education, 3, 351-361. doi:10.12973/ejmste75414

Pettit, R. S., Homer, J., & Gee, R. (2017). Do enhanced compiler error messages help students? In M. E. Caspersen & S. H. Edwards (Eds.), Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education—SIGCSE’17 (pp. 465-470). doi:10.1145/3017680.3017768

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, 211-227. doi:10.1002/sec.3730660207

Price, T. W., Dong, Y., & Lipovac, D. (2017). iSnap: Towards intelligent tutoring in novice programming environments. In M. E. Caspersen & S. H. Edwards (Eds.), Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education—SIGCSE’17 (pp. 483-488). New York, NY: ACM Press. doi:10.1145/3017680.3017762

Qian, Y. (2018). Understanding and addressing misconceptions in introductory programming: A data-driven approach (Doctoral dissertation). Purdue University, West Lafayette, USA.

Qian, Y., & Lehman, J. (2017). Students’ misconceptions and other difficulties in introductory programming: A literature review. ACM Transactions on Computing Education, 18, 1:1-1:24. doi: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, 203-221. doi: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, 60-67. doi: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, 37-64. doi:10.1007/s40593-015-0070-z

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

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%27+Programming+Knowledge#0
Shulman, L. (1986). Those who understand: Knowledge growth in teaching. *Educational Researcher, 15*, 4-14.

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

Simon, (2011). Assignment and sequence: Why some students can’t recognise a simple swap. In A. Korhonen & R. McCartney (Eds.), *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (pp. 10-15). New York, NY: ACM Press. doi:10.1145/2094131.2094134

Sirkia, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In M.-J. Laakso & Robert McCartney (Eds.), *12th Koli Calling International Conference on Computing Education Research* (pp. 19-28). New York, NY: ACM Press. doi:10.1145/2401796.2401799

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*, 5-23. doi:10.2190/2XPP-LTYH-9NQ-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

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

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education, 13*, 1-31. doi: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*, 1-64. doi:10.1145/2490822

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

Taber, K. S. (2014). Alternative conceptions/frameworks/misconceptions. In R. Gunstone (Ed.), *Encyclopedia of science education* (pp. 1-5). Dordrecht, The Netherlands: Springer. doi:10.1007/978-94-007-6165-0_88-2

Vosniadou, S. (1994). Capturing and modeling the process of conceptual change. *Learning and Instruction, 4*, 45-69. doi: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 & Francis.

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

Webb, M., Davis, N., Bell, T., Katz, Y., Reynolds, N., Chambers, D. P., & Sysło, M. (2017). Computer science in K-12 school curricula of the 21st century: Why, what and when? *Education and Information Technologies, 22*, 445-468. doi:10.1007/s10639-016-9493-x

Xu, S., & Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering, 29*, 360-384. doi: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. In J. Vahrenhold & E. Barendsen (Eds.), *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 92-95). New York, NY: ACM Press. doi:10.1145/2978249.2978264

**Author Biographies**

**Yizhou Qian** is an assistant professor of educational technology at Jiangnan University, China. His research focuses on computer science education, student misconceptions in introductory programming, and data-driven programming learning systems.

**James D. Lehman** is professor emeritus of learning design and technology at Purdue University. His research interests include technology integration in STEM education, e-learning, educational multimedia, and computer science education.