Journal of Information Systems Education

Volume 34
Issue 1
Winter 2023

Teaching Tip

An Example-Based Instructional Method to Develop Students' Problem-Solving Efficacy in an Introductory Programming Course

Pratibha Menon

Recommended Citation: Menon, P. (2023). Teaching Tip: An Example-Based Instructional Method to Develop Students' Problem-Solving Efficacy in an Introductory Programming Course. *Journal of Information Systems Education*, 34(1), 1-15.

Article Link: https://jise.org/Volume34/n1/JISE2023v34n1pp1-15.html

Received: November 23, 2021
Revised: January 22, 2022
Accepted: May 4, 2022
Published: March 15, 2023

ISSN: 2574-3872 (Online) 1055-3096 (Print)

Journal of Information Systems Education, 34(1), 1-15, Winter 2023

Teaching Tip
An Example-Based Instructional Method to Develop Students' Problem-Solving Efficacy in an Introductory Programming Course

Pratibha Menon
Department of Computer Science and Information Systems
Pennsylvania Western University
California, PA 15419, USA
menon@pennwest.edu

ABSTRACT

This paper introduces a teaching process to develop students' problem-solving and programming efficacy in an introductory computer programming course. The proposed teaching practice provides step-by-step guidelines on using worked-out examples of code to demonstrate the applications of programming concepts. These coding demonstrations explicitly teach the systematic approach and strategies required to develop a programming solution. Each code demonstration is then followed by the instructor assigning similar practice problems to build learners' awareness of the programming process and problem-solving techniques. Every successful attempt of the practice exercise by a student exemplifies their efficacy in applying the programming process and developing solutions using the instructor's strategies. Finally, through regular and structured feedback, the instructor gives learners insight into their performance in completing various steps of the programming process. This paper provides guidelines for creating and using code demonstrations, practice exercises, and rubrics for structured feedback in an introductory programming class. An end-of-course survey was employed to compare students' reported self-efficacy and their actual programming and problem-solving efficacy, based on their completion rates of the practice activities.

Keywords: Self-efficacy, Computer programming, Feedback, Exercises, Code demonstrations

1. INTRODUCTION

Computing and information systems undergraduate students generally regard the compulsory introductory programming courses as complex. A significant number of students drop out, leading to attrition during the first and second years (Beaubouef & Mason, 2005; Kinnunen & Malmi, 2006). Teaching students to write computer programs in an introductory course transcends the programming languages and tools they may use to develop code. Writing code requires students to develop the cognitive skills necessary to monitor their programs and apply effective strategies to fix errors and solve problems.

Learning how to write well-documented, error-free, and efficient programs, as with any expertise, requires students to develop an awareness of their thoughts and actions and to regulate their efforts to meet their learning goals. Studies suggest that the ability to self-monitor and formulate explanations independently improves problem-solving skills and self-efficacy (Cleary & Zimmerman, 2012; Cleary et al., 2006). Self-efficacy, which influences a learner's regulation of their learning process, refers to people's judgments of their ability to organize and execute the courses of action required to attain the desired performance. Self-efficacy enables an individual with prior knowledge and skills to take the action necessary to complete a task (Bandura, 2012). Additionally, a previous study indicates that individuals' beliefs strongly influence their behavior, and that knowledge, skills, and prior attainment alone may not be strong indicators of subsequent achievements (Pajares & Miller, 1994).

This paper focuses on implementing a process guided by self-efficacy theories to teach the fundamentals of procedural programming using the Java programming language. Appendix A outlines the course objectives and topics covered in the 15-week course studied in this paper. A key feature of the proposed instructional design is its incorporation of an example-based learning approach to teach the code development process and the problem-solving strategies required to develop robust programming solutions.

In the proposed approach, students learn problem-solving methods and strategies by observing the instructor explain worked-out examples. The instructor introduces key programming concepts through code demonstrations and systematically models the programming process and problem-solving strategies. These include translating problem requirements into code components and sequence, tracing the variables, altering code, and fixing errors. Each code demonstration is accompanied by a group of pre-written practice activities that target some of the debugging problems commonly encountered by students. These activities help students repeatedly practice valuable strategies to troubleshoot and fix errors. Eventually, they transition to independently

writing more extensive programs similar in scope and scale to the examples demonstrated through the code demonstrations.

Research demonstrates that instruction using worked-out examples is generally more effective and efficient for novice learners than instruction consisting primarily of problem-solving exercises (Huang, 2016; Renkl, 2011; van Gog & Rummel, 2010). For example, a traditional approach to instruction may involve lectures that introduce a set of concepts, followed by assignment problems containing a given set of values, conditions, and a goal statement. However, assigning problems without explaining the details of the problem-solving procedure leads novices to resort to weaker problem-solving strategies. For example, without knowing the optimal approach an expert would apply to solve problems, a beginner may use means-ends analysis, searching repeatedly and inefficiently for the operations needed to reach the goal. Even though less efficient strategies may allow learners to solve the problem eventually, they do not contribute to learners building a cognitive schema for solving similar problems. Such a schema can extend beyond the specific problem-solving procedure to enable the reuse, adaptation, and transfer of problem-solving skills to newer problems (Cooper & Sweller, 1987; Paas, 1992).

Prior studies on self-efficacy show that vicarious learning experience gained through the observation of others' modeled performance provides learners with the aspiration to attain a given level of performance (Bandura, 1996). Therefore, by modeling an instructor's programming practices through worked-out examples, students could learn efficient and systematic problem-solving methods and self-monitor their problem-solving strategies to improve their chance of success. The cognitive apprenticeship model proposed by Collins et al. (1989) includes six teaching methods—modeling, coaching, scaffolding, articulation, reflection, and exploration—that make explicit the expert's tacit knowledge for students from which to learn. The cognitive and metacognitive aspects of such a model deal with the processes and strategies used to solve problems, which are helpful in situations that require students to extend their knowledge to novel situations and complex problems. A study by Loksa et al. (2016) indicates that explicit instruction on programming problem solving has improved students' programming self-efficacy in a controlled study.

While modeling a problem-solving process allows students to emulate the instructor's cognitive schema and set efficacy expectations, completing many practice problems could strengthen students' problem-solving abilities. Furthermore, personal mastery experiences also influence students' perceived self-efficacy (Bandura, 2012). Repeated success in attempting many practice exercises improves mastery and, therefore, efficacy expectations, which in turn could reduce the negative impact of occasional failures. Furthermore, presenting learners with an example followed by similar practice problems is more effective than providing just the examples (Renkl, 2011; Renkl, 2014; van Gog et al., 2011). Therefore, the instructional approach proposed in this paper pairs detailed coding demonstrations with various practice problems that allow students to apply problem-solving strategies repeatedly and, as a result, improve their expectation of efficacy.

The rest of the paper is organized as follows. Section 2 explains our teaching process by providing guidelines on developing instructional components such as code demonstrations, practice exercises, structured feedback, and question and answer (Q&A) sessions. Section 3 provides the author's guidance on developing code demonstrations and different kinds of practice exercises. Section 4 discusses the results of an end-of-the-course survey used to measure the students' perceptions of their self-efficacy in solving problems and completing different types of practice exercises. Section 5 discusses the problem-solving and programming abilities displayed through students' completion rates of practice exercises. Section 6 discusses how various attributes of the instructional design could have impacted students perceived self-efficacy and observed problem-solving efficacy. Finally, Section 7 concludes the paper by summarizing the teaching practice and potential investigations and improvements for the future.

2. THE TEACHING PROCESS

The design of the teaching process adopted in this study assumes that learning computer programming occurs as a cyclical exchange of knowledge and information between the learner and an external learning environment. The learner's interaction with the learning environment is assumed to occur in two ways: 1) between the learner and the teacher and 2) between the learner and external learning tools such as an integrated development environment (IDE). These interactions are called teacher-practice and teacher-modeling, respectively (Laurillard, 2012). Figure 1 depicts the instructional components of the teacher-practice and teacher-modeling cycles and the association between the two.

The teacher-practice cycle represents the teacher's role in scaffolding the programming and problem-based learning process. The teacher designs programming and problem-solving exercises relevant to the content and the student's level of knowledge and expertise. By revealing the teacher's practice through suitable instructional activities, learners obtain the means to build an external representation of their learning. The teacher may also provide means for students to discuss, ask questions, and clarify their understanding. The teacher-practice cycle reflects the portion of the instructional design to be covered through class activities, lectures, and discussions in the teacher's presence.

The teacher-modeling cycle involves a modeling environment in which students complete thoughtfully designed tasks to practice their programming knowledge and obtain meaningful feedback. The modeling environment models the learning task so learners can observe the result of their actions and compare it to the intended results. Such a modeling environment enables learning if students can complete the tasks and interpreting the feedback.

In a typical programming class, the IDE takes the role of the teacher by providing students with immediate feedback on the correctness of their code through error and exception reports. Other tools may include auto-graded online quizzes that provide immediate feedback on students' conceptual understanding of programming concepts. Students complete the learning activities associated with the teacher-modeling cycle at home. The online learning tasks are hosted in a learning management system, and students are provided access to the IDE through a virtual desktop.

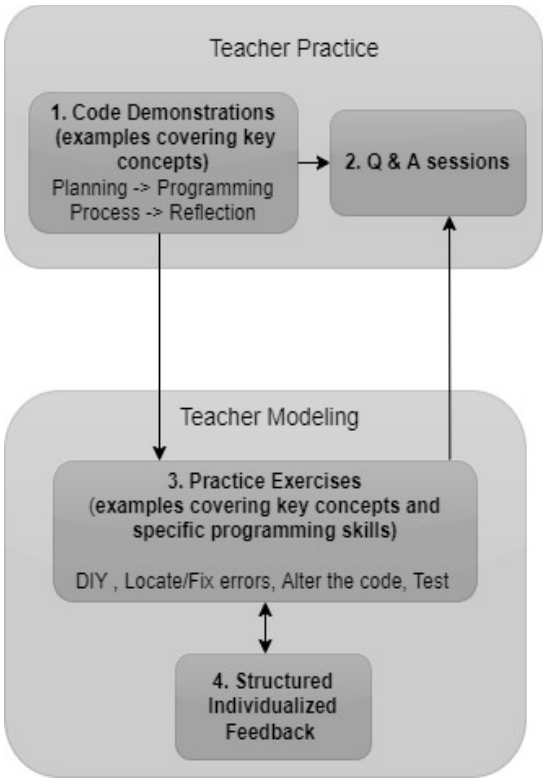


Figure 1. Structure of the Instruction Process

The instructional design explained in this section illustrates the knowledge transfer between teacher and learners through code demonstrations; practice exercises; Q&A sessions; and regular, structured feedback.

Code demonstrations, which form one of the critical activities of the teacher-practice cycle, exemplify the teacher's practice of writing good programs. Practice exercises allow students to emulate the teacher's way of writing programs and solving problems. Students write programs independently and submit the code for review and feedback by the teacher. Q&A sessions allow students to discuss their code, clarify their understanding, and check their programming solutions with the teacher's assistance. Q&A sessions are also used to review assignments and online quizzes.

The practice exercises adopted during the teacher-modeling cycle focus on developing the intuition and thought process required to create a programming solution. Students subsequently come to realize that coding should be preceded by thoughtful analysis of the problem and the solution's requirements. Many different learning tools currently exist to introduce software development to beginners without having them use a strongly typed programming language or a text-based programming method. For example, block-based and visual programming approaches could teach students to create computer programs without writing much syntax. However, a study by Weintrop and Wilensky (2017) that compared the use of text-based and block-based programming methods did not find any difference in students' programming confidence or enjoyment. The study also found that, compared to the group that used the block-based method, students who used the text-based programming method viewed their programming experience as closer to what professional programmers do and more effective at improving their programming ability.

Using end-user development tools, such as low-code development platforms, spreadsheets, or tools requiring limited scripting, is another approach to allow students with little or no prior programming background to develop functional software solutions. End-user development tools decrease the learning effort required to develop applications (Fischer et al., 2004). By using the end-user development approach, developers can utilize the features of the development tool to assemble a solution quickly. These end-user tools, however, conceal several fundamental aspects of learning how to program, such as knowing about data types, tracing variables' states, and understanding how the program compiles and executes. As a result, by using end-user development tools, students may not receive sufficient instruction and practice to develop their programming intuition. This study therefore introduces basic programming using the Java programming language and an IDE, such as Eclipse, to help students build and test their code.

Instructional Activities—Teacher-Practice Learning Cycle			
	Forethought	Performance/ Programming Process	Self-Reflection
Code Demonstrations	Problem analysis, solution planning, reviewing test plans	Choosing constructs/concepts, specifying variables, identifying sequence, tracing variables, altering code, running tests	Evaluating style, practices, and errors
Q&A Sessions	Task planning, goal setting to improve the learning process	Discussions on identifying and correcting errors; adopting best practices	Choosing practice materials to strengthen practice

Table 1. In-Class Instruction—Code Demonstrations and Q&A Sessions

2.1 The Code Demonstrations

The teacher-practice component of the teaching process consists of the code demonstrations through which an expert instructor models program development process. The instructor delivers the code demonstration in three consecutive phases: the forethought, the performance, and the reflection phases. These three steps align with the self-regulated learning (SRL) model identified by Zimmerman (2009). Table 1 explains how instruction using code demonstrations and Q&A sessions could capture the SRL process's forethought, performance, and self-reflection phases. Each code demonstration begins with the instructor describing their forethought on ways to approach the problem, followed by a performance phase consisting of task analysis, code development, execution, and testing. Finally, before concluding a code demonstration, the instructor reflects on the coding process and the solution. The sample code used

for the demonstration contains extensive documentation and comments that students can refer to later.

The code demonstrations introduce programming concepts by discussing examples of their applications. The instructor's explanations contain the rationale for using one or more programming constructs and ways to apply these constructs to solve a given problem. For example, the instructor may introduce the topic of loops by demonstrating applications that require the repetition of code segments. Furthermore, the explanation explores a range of loops to choose from, such as "do-while," "while," or "for" loops. In some cases, the instructor may configure all three types of loops to solve a given problem and compare their differences.

Aside from teaching students how to pick and configure appropriate programming constructs, the code demonstrations are also used to teach critical troubleshooting skills. For example, tracing the code and modifying the results are vital skills for debugging programs. Students are also taught to adapt pre-existing solutions to solve new but related problems.

By making the program development process detailed and explicit, instructors provide students with a language to discuss and explain the various steps required to develop a program. The worked-out examples in the code demonstrations apply the general programming process illustrated in Figure 2 to create programs. These programs may use different programming concepts and constructs, such as variables, data types, expressions, statements, decision structures, loops, and methods. In addition, the instructional materials and practice activities emphasize applying the general programming process to develop and debug the programs.

While a walk-through of the programming process is at the heart of the code demonstration, it is essential to precede this process with the instructor's forethought and to conclude it by reflecting on what has occurred. The forethought phase includes approaches to analyzing the problem and planning the solution, while the reflection phase includes evaluating types of potential errors and suitable writing styles. Instructors routinely discuss the forethought and self-reflection phases of learning during the Q&A sessions to guide students through the performance phase they must complete independently while attempting the programming tasks. A typical code demonstration takes approximately 45 minutes of class lecture time.

2.2 The Q&A sessions

Q&A sessions are integral to the teacher-practice learning cycle. They are 20- to 30-minute sessions reserved for discussions and further clarification of the concepts explored during the regular class session. During the Q&A sessions, the instructor clarifies any misconceptions or problem-solving difficulties students have experienced while completing learning activities. The instructor may also discuss the graded assignments and some of the common errors and misconceptions that were evident in student submissions.

2.3 The Practice Exercises

As illustrated in the instructional process depicted in Figure 1, the teacher-modeling phase includes several practice exercises. Students apply the teacher's program development practices, previously explained through the code demonstrations, by attempting similar practice exercises. Table 2 lists the different categories of practice exercises included in the course content. These exercises are completed as homework assignments. For example, the do-it-yourself (DIY) exercises are significant problems similar in scope and size to the code-demonstration problem. They require students to follow most of the steps in the programming process depicted in Figure 1. In this type of activity, students analyze a problem, identify the potential solution, implement it, and test the answer. Observing the sample code provided during the code demonstration allows students to recollect and emulate the instructor's practices to write the code by themselves using an IDE. The DIY activities also advise students to analyze the problems, write extensive comments, and build their code incrementally.

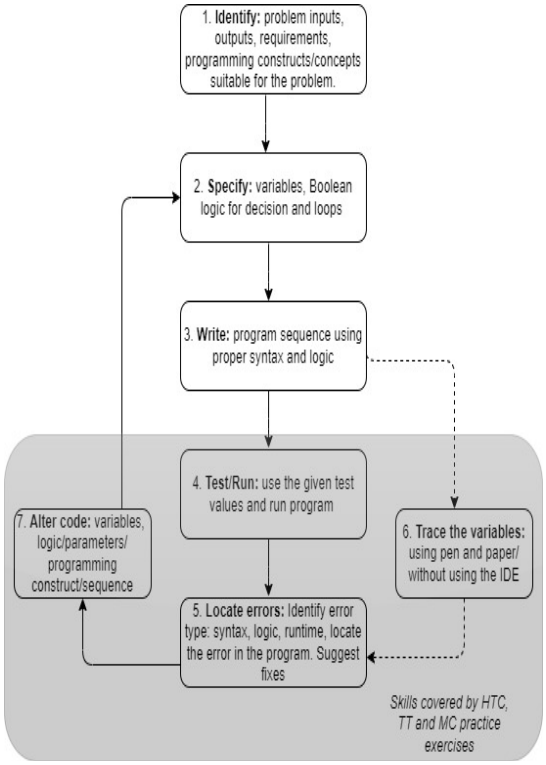


Figure 2. The Programming Process

In addition to the DIY activities, the practice exercises also include shorter skill-building exercises such as hack the code (HTC), test tube (TT), and messed-up code (MC). These skill-building activities help students practice and become comfortable detecting and correcting logical, syntactical, and runtime errors. For example, the MC contains one or more errors that students must identify and correct. HTC is an activity in which students are required to alter a pre-written code's logic to obtain the required outputs.

The MC and HTC activities encourage students to feel comfortable experimenting with their code. Another activity that enables students to solve problems by experimentation is the TT activity. This activity requires students to test a given code by tracing the variables using pen and paper. These activities allow students to develop the necessary troubleshooting skills and a sound conceptual understanding as they learn to write programs. Appendix B presents samples of each practice activity and the approximate time it would generally take a student from an introductory programming class to complete. For example, a typical short practice exercise

would take approximately 20 minutes, while DIY exercises take about an hour to complete.

	Targeted Practice for the Teacher-Modeling Learning Cycle
DIY	Problems like the ones from the code demonstration. Give a mini case; write the problem requirements; develop sequence/logic; and apply strategies such as testing, adjusting/altering code, tracing variables, and fixing errors. Compile using IDE.
Testing	Test a given code by varying the list of possible inputs. Identify the correctness of the code and the exceptions it might produce. Test using IDE.
Fix Errors (Messed Up Code)	Analyze the sequence, logical construct, and its parameters on an errored code and suggest possible fixes. Fix and mitigate errors using the IDE.
Alter the Code (Hack the Code)	Experiment with a given code to produce a different set of outputs (including errors). Develop the ability to draw upon past code examples and predict the outputs. Produce a modified code using the IDE.
Tracing Variables (Test Tube)	Using paper and pencil, trace the state of variables in a code without using an IDE. This helps students to comprehend a given code.

Table 2. Different Types of Practice Exercises

2.4 Structured Feedback

After students complete and submit the assigned practice problems, the instructor provides grades and feedback at the beginning of the following week. The instructor reviews the submissions every week and provides feedback to each student using a rubric shown in Appendix C. The instructor informs the students of the exact programming process steps they have completed and the ones that might be causing errors in the solution. The feedback identifies the stage of the programming process at which the student could have made an error, although it does not explicitly state how to correct the mistakes.

Students can follow the instructor's feedback to correct and resubmit the solution before a final submission deadline. The contents of a module are covered for roughly a month. The hard deadline for submitting all exercises for a module is typically the last Sunday of the month. Students should complete all DIY activities before the hard deadline, after which the activities are graded. Based on the rubric, the feedback indicates how well a student has followed various steps in the programming process to develop solutions for the DIY problems. By using the same feedback rubric for all the DIY exercises, students can monitor their progress in mastering the programming process and problem-solving strategies throughout the semester.

3. GUIDELINES FOR DEVELOPING COURSE CONTENTS

The proposed teaching process requires instructors to plan the delivery of the code demonstrations and finalize the composition of the practice exercises. Keeping a consistent pattern of explanations in the code demonstrations could help students set their expectations of class time. Repetition of the explanation pattern also helps students internalize the instructor's approach to solving programming problems.

3.1 Developing the Code-Demos

As illustrated in Table 1, the code demonstrations consist of three phases: the forethought, the performance, and the reflection. Therefore, the steps listed below could be used as a guide for instructors to develop code demonstrations that cover all three phases:

1) Provide a detailed explanation and analysis of the problem statement to identify the functional and data requirements. Refer to past examples that have similar solutions.
2) Identify ways in which the given problem may differ from the past examples.
3) Identify problem requirements by mapping the information given in the problem to the input data and identifying the required outputs.
4) Write comments before writing the code by listing the problem's inputs and the expected results.
5) Identify some of the critical programming constructs to solve the problems.
6) List the required variables and their data types that will be used to store the input and output values.
7) Declare the required variables and their data types by writing Java statements.
8) Write the code sequence containing the correct logic and syntax using an IDE.
9) Test the code incrementally.
10) Identify and fix any syntactical, logical, and runtime errors.
11) Trace the values of variables by stepping through each line of the program.
12) Alter the code in at least two different ways to obtain different program outcomes.
13) Comment on acceptable coding practices and writing styles relevant to the problem.
14) Reflect on some of the common errors and challenges commonly encountered by learners while solving a similar problem.
15) Recommend ways to improve the problem-solving and programming skills required to solve similar problems.

Steps 1, 2, and 3 consist of activities required to plan the code, and Steps 4, 5, and 6 involve identifying the problem requirements. After identifying the program requirements, the instructor develops the program sequence using the variables and logical constructs and describes how to test and develop the program incrementally. Students also observe how the instructor applies techniques such as tracing the variables or printing out the variables' values to incrementally build their code during the code demonstration. Steps 7, 8, and 9 translate the problem requirements into code using the correct syntax and sequence. Thereafter, Steps 10, 11, and 12 teach students to test and develop their code incrementally. In these two steps, the instructor explains the strategies that students could use to fix errors in their code. After demonstrating how to develop and test the program, the instructor moves to the self-reflection phase of the code demonstration, listed in Steps 13, 14, and 15.

All the example programs discussed as code demonstrations incorporate all the instructional steps listed above. In fact, an instructor could use these steps to record video lectures for online course delivery.

3.2 Developing Practice Exercises

Just as the code demonstrations have a predictable pattern of explanation, the practice exercises also have predictable composition and submission patterns. The entire course is divided into four broad topics or modules. Appendix D presents the classification of the practice activities into four assignment modules, each of which requires a time frame of three to four weeks to cover the contents thoroughly. Module 1 introduces the concepts of variables and data types as well as simple input and output methods. Students write a simple program to input and output values and perform simple print statements and arithmetic operations. Module 2 introduces Boolean expressions and problem-solving using different decision structures using "if… else" statements. Module 3 covers problem solving using different types of loops. Module 4 teaches students to modularize their programs using methods.

Each module builds upon the prerequisite concepts covered in previous modules. Moreover, each module consists of a set of pre-planned practice exercises. Students complete the practice exercises independently, and these exercises are therefore crucial to developing conceptual knowledge and the ability to write error-free programs. These practice exercises could be graded assignments throughout the course, in which case students must complete all the practice exercises as graded assignments.

The number and types of practice exercises in a module depend on the number of concepts covered in that module. Additionally, the number of practice exercises is constrained by the time required for students to complete them. The following criteria were used to create the practice exercises:

1) Each problem and solution discussed in the code demonstration will be followed by at least two similar DIY problems.
2) TT, HTC, and MC activities are incorporated into the module assignments.
3) The exercises will require students to test their code and submit error-free solutions.
4) The size and scope of the DIY activities will be such that it would take a student, on average, one hour to fully complete, test, and document their code.
5) The scope and size of the practice exercises will be such that it would take an average of 20 minutes for a student to complete them.
6) The total number of practice and DIY exercises is limited by the estimated weekly time a student would spend on completing them.

Many of the exercises require students to perform more than one step in the programming process. On the one hand, for example, almost all the targeted practice activities require students to test the code by running it using an IDE. On the other hand, the DIY activities require students to follow all the main steps of the programming process systematically and, optionally, to trace the variables and alter the code. The HTC, TT, and MC activities require students to analyze or change a given code. By contrast, the DIY activities ask students to identify the problem requirements and the program sequence and to develop and test the code. The DIY activities are similar in scope and complexity to the problems discussed during in-class code demonstrations. On average, a student would take approximately an hour to fully complete the writing and testing of the program. Students attempt all the assignments and practice activities independently, although they could use Q&A sessions to clarify their understanding of the problems from the instructor.

Appendix B offers a sample set of practice activities covered for each topic/module during a 15-week semester. The instructor had planned and prepared these practice exercises at the beginning of the semester. The chart in Appendix C maps each practice exercise to the set of programming skills it could help students develop. The names of the problems are chosen to describe the application that the problem intends to solve using programming solutions. For example, a problem in Module 1 on data types is named "FlooringCost" and not "DataTypes 5." By giving the problem a meaningful name that relates to its application, the instructor can refer to it later in the course to show how it may be extended using more advanced programming constructs. It also becomes easier to discuss programming solutions if students can easily recollect the solution patterns and apply them to similar problems.

4. STUDENT PERCEPTIONS

The impact of the proposed instructional design was studied in a 15-week introductory Java programming course for a computer information systems program at a public university. Twenty-one students had enrolled in the course, 19 of whom voluntarily participated in an anonymous post-course survey. The survey measured students' perceived self-efficacy having completed the course.

4.1 Value of the Learning Activities

Of the 21 students who attended the course, 19 volunteered to participate in the non-mandatory end-of-course survey. Since the practice exercises constituted a large part of the instruction, the end-of-course survey included questions on the students' perceptions of the value of various types of practice/learning activities (such as the DIY, HTC, TT, and MC). The survey asked students to rate the value of each learning activity on a Likert scale from 0–4, where 0 represents "Very Much Disagree," and 4 represents "Very Much Agree." Table 3 depicts the results of the post-course survey, showing that most students felt the learning activities were valuable in the course. The survey question is stated in the top row of Table 3. The author of this paper created the survey instrument used in this study.

The DIY activities received slightly more neutral responses than the others. The class instructor observed that the DIY activities appear to be more challenging than the practice activities because they require students to apply all the steps of the programming process independently. The DIY problems cover more programming concepts than the targeted exercises.

Question: How valuable were the following learning activities in developing your programming skills in this course?					
	Very Much Disagree	Disagree	Neutral	Agree	Very Much Agree
Q&A sessions	0	0	1	11	7
Fix Errors—Messed-Up Code activities	0	0	1	12	6
Write/alter code—Hack the code activities	0	0	1	11	7
Trace variables (Test Tube)	0	0	1	7	11
Develop code (DIY activities) using Eclipse IDE	0	0	4	13	2

Table 3. Student Response Distribution on the Value of Various Learning Activities in Developing Programming Skills

Additionally, the feedback detailed the DIY problems but only offered commentary on the correctness of the shorter practice problems. Based on the instructor's observation, most of the Q&A questions concerned the initial steps of the programming process required to develop sequence and logic. Most of the questions raised by students pertained to identifying programming requirements for a given problem, such as the data types, logical constructs, Boolean expressions, and method parameters.

4.2 Perceptions of Self-Efficacy

The post-course survey also contained questions on self-reported measures of efficacy. Table 4 lists the survey questions on students' perceived self-efficacy to complete the various steps of the programming process illustrated in Figure 2. In addition, the survey asked students to rate their confidence and abilities related to various steps of the programming process on a 0–4 Likert scale, where 0 represents "Very Much Disagree" and 4 represents "Very Much Agree."

Table 4 indicates that most students reported higher confidence in their ability to troubleshoot errors, and most students felt confident experimenting with their code. Practice activities such as TT, MC, and HTC are exercises requiring students to identify and fix errors to build troubleshooting skills. Students experiment with various data inputs in many of these exercises and alter the code to meet the problem requirements. Student responses in the survey indicate an overall positive efficacy in eliciting program requirements before writing the code using Java syntax. Five out of 19 students, however, gave a neutral response on their ability to elicit program requirements. One possible explanation for the neutral responses is that no specific skill-building activities provided students with focused practice in identifying problem requirements. For example, the instructor frequently had to guide students in translating the given problem statements into the sequence of operations and statements. Even though students could refer to worked-out examples to observe the solution patterns, they still had to engage in a deliberate thought process to tailor their solutions sufficiently for the assigned problem. These actions include mapping the given data and the required outputs into relevant variables and data types and identifying the essential operations, structure, flow control elements, and method parameters. The instructor also observed that many students repeatedly skipped planning their code and structuring their solution before starting to type their code in the IDE. As a result, students did not identify the correct requirements, resulting in too many logical errors in the answers.

Readers should note that students' reported self-efficacy may not accurately reflect their actual ability to complete programs and solve problems. A cognitive bias called the Dunning–Kruger effect could lead poor performers to be overconfident in their skills and top performers to underrate themselves (Dunning, 2011). Nevertheless, students' perceptions of self-efficacy could be an essential motivating factor for them to persist with computer programming courses in the future.

This study does not have a mechanism to infer how accurately students calibrate their perceived self-efficacy based on their actual performance, which could be observed through grades and instructor feedback. The anonymous nature of the self-efficacy survey prevents matching a student's performance with the reported self-efficacy measurements. However, collecting the students' identities with their reports of self-efficacy could have motivated them to misreport their true perceptions of self-efficacy in the interests of social acceptability. Furthermore, the self-efficacy survey is administered at the end of the course. Therefore, student responses could have resulted from their cumulative problem-solving and programming experience throughout the course duration.

5. STUDENT PERFORMANCE

This study took place in a live classroom in which an instructor needed to keep track of the learning outcomes and the class performance in completing the assignments. The task completion percentage for every assignment activity submitted by each student in the class was recorded to measure actual programming and problem-solving efficacy. Appendix C tabulates all the assignment activities for the entire semester.

5.1 Practice Exercises Completion Percentages

The task completion percentage is the percentage of assigned practice activities that a student fully completed without errors. Assignment questions that were only partially completed or had errors were not counted while calculating task completion percentage. The task completion percentage was not collected for Module 1 since this period coincided with the add/drop period of the semester. Students who joined late in the class got caught up with the course materials by the beginning of Module 2. Students have received help from the instructor to complete the practice activities via the structured time available during the Q&A sessions. The Q&A sessions encourage students to use the vocabulary and the steps of the programming process during the class discussions. Table 5 shows the mean value of task completion percentage for assignment questions in each module. The mean value of assignment completion percentages decreased for each subsequent module. It is also observed that the standard deviation increased over the semester, indicating that the gap between the stronger and weaker performers increased over the semester.

Question: For each of the following questions, please rate your perceived efficacy in completing various steps of the programming process.					
	Very Much Disagree	Disagree	Neutral	Agree	Very Much Agree
I feel confident to experiment with my programs.	0	0	1	8	10
I feel confident that I can correct programming errors.	0	0	0	9	10
I feel that learning how to program has improved my problem-solving skills.	0	0	3	7	9
I know how to read a given problem and deduce the sequence of operations and statements required to write a programming solution.	0	0	5	10	4
I know how to correctly identify the statements required to write a programming solution.	0	0	5	9	5

Table 4. Student Response Distribution on Indicators of Students' Programming Self-Efficacy

Assignment Completion Percentage			
	Module 2	Module 3	Module 4
Total number of Assignment problem	13	15	11
Min and Max number of concepts in a problem in an assignment	Min: 4 Max: 6	Min: 6 Max: 8	Min: 7 Max: 9
Number of DIY activities assigned	6	8	6
Number of skill building (TT, HTC, MC) activities	7	7	5
% Task Completion: Mean	83.1	79.7	73.7
% Task Completion: Std. Dev	25.7	27.5	31.5

Table 5. Assignment Completion

5.2 Assignment Complexity

The number of programming constructs that a student must apply to solve a problem point could be a proxy measure for task complexity. Each assignment problem is tagged with the programming concepts essential to developing the solution. Each example problem discussed in the code demonstrations typically involves many programming concepts. Each practice problem that follows the code demonstration also involves more than one concept. The following concepts were covered in code demonstrations and practice exercises: 1) variables, 2) data types, 3) operations, 4) expressions, 5) statements, 6) loops, 7) decisions, 8) methods, 9) parameters/arguments, and 10) return values.

Table 5 indicates that the minimum and maximum number of concepts covered in a problem increased as the semester progressed. For example, Module 1 covered only four concepts, but there are at least seven concepts per problem from Module 4 onwards. The maximum number of concepts in a Module 2 problem never exceeded six, but the maximum number of concepts in a Module 4 problem was nine. The DIY exercises typically covered more concepts than the shorter exercises. Module 4 exercises covered and built upon many of the concepts from Module 2. The module-level task completion decreased for Module 3 and again for Module 4. It also appeared that the standard deviation was higher for Module 4 exercise completion. This indicates that the conceptually complex Module 4 assignments must have been challenging for students to complete fully.

6. DISCUSSION

The post-course survey results revealed that the majority of students positively agreed on the value of most of the practice exercises. Most students also agreed on their self-efficacy to develop computer programs. While the instructor is responsible for developing students' programming efficacy by creating suitable course contents, it is equally important to do so in a way that supports students' perceived self-efficacy. At the same time, accurate feedback on student performance is essential for the student to calibrate their perceived self-efficacy to their actual problem-solving and programming efficacy. A study by Moores and Chang (2009) suggests that self-efficacy could be positively correlated to true efficacy; still, incorrect perceptions of self-efficacy could lead to overconfidence and a subsequent drop in future performance.

Post-course survey results indicate that more students agreed or strongly agreed with the task value of the targeted practice exercises, such as the TT, MC, and HTC, than they did for the more extended DIY activities. Successful completion of the targeted practice activities could have helped reinforce the perceived self-efficacy of the student more frequently. These short activities had students work on pre-written code. They had to correct errors, predict the outputs by tracing the variables, or alter the logic and sequence to obtain different results. By contrast, the more prolonged DIY activities required students to identify the problem requirements independently, evaluate multiple solution patterns, and complete all the steps of the programming process. Even though the code demonstrations

had explained how the instructor extracted the programming requirements from the problem statements, there were no additional exercises that specifically targeted requirement elicitation or focused on identifying the sequence and operations from reading problem statements. Future iterations of the course design could incorporate practice activities to help students identify the problem requirement, logic sequence, and correct operators prior to writing their code.

Developing an error-free solution for a DIY activity required students to invest more time and effort into creating and testing a programming solution than were required for the shorter practice activities such as the TT, HTC, and MC. As a result, feedback on the DIY activities was also more extensive. The feedback provided for the DIY activities objectively mapped the correctness of the program for various steps of the programming process. Although students received positive feedback for getting their programs correct, they also received commentary on the improvements they could make to their incorrect, inaccurate, or incomplete solutions. The DIY exercises, as a result, made it possible for students to reflect upon their efficacy extensively.

The extent to which students were accurate in calibrating their self-efficacy based on the detailed feedback they received for their DIY solutions could not be inferred due to the anonymity of student survey responses. However, what could be measured was the efficacy of the class as a whole. Students' ability to complete the programming assignments fully could be observed by collecting the assignment samples. Table 6 indicates that the aggregate task completion percentage declined in the later modules, possibly due to the complexity of the topics. The complexity of the problems could have made the solutions more prone to errors in the latter part of the course. These errors could have influenced the survey that collected data on perceived self-efficacy during the final week of the course.

While students would have inferred the increasing complexity of problems in the later modules, the feedback provided by the instructor did not explicitly relate students' performance to the complexity of the problem, nor did the feedback indicate students' progression in complex problem-solving capabilities over the semester. Future iterations of the instructional design could benefit from an enhanced feedback method that informs students about the complexity of the tasks they have completed during the course. In addition, providing each student with a progress trajectory that tracks their performance according to the complexity of the problems solved could help students calibrate their perceived self-efficacy with their actual problem-solving and programming efficacy.

The instructional design proposed in this paper stresses developing a set of practice activities and code demonstrations. Content development is always a time-consuming process. However, creating a template for the learning activities and repeating the problem templates for different learning units may save the instructor some time while developing content. For example, the short practice problems were restricted to the TT, MC, and HTC types. The modules had a predictable mixture of short practice activities and more extended DIY exercises.

One of the challenges of applying the instructional design proposed in this course is the creation of practice exercises. Repetition of the same pattern of practice exercise, such as the DIY, TT, MC, and HTC, makes it easier to generate assignment questions across different learning units. In addition, developing question templates and a sharable repository of practice exercises will make it easier for novice instructors to adopt the instructional method proposed in this paper.

7. CONCLUSIONS

Computer programming courses are valuable for information systems curriculums, as they provide a structure to analyze problems and develop solutions. This study proposes an example-based instructional method to teach programming and problem-solving strategies by creating code demonstrations and practice problems. The instruction provided by the code demonstrations and the programming and problem-solving steps targeted by the practice problems capture the forethought, the programming process, and the self-reflection required to develop programming solutions. This paper provides instructors with a step-by-step guideline to build code demonstrations and practice exercises such that students develop the skills necessary to complete various steps of the programming process. Just as the examples of code demonstrations provide students with program structure, the practice exercises offer them examples of ways to test, debug, and alter solutions. In addition, each successful attempt of the practice exercise helps exemplify students' programming self-efficacy. The worked-out examples in the code demonstrations, practice problems, Q&A sessions, and structured feedback that inform a student's mastery of the programming process are repeated for every unit of study throughout the semester.

A post-course survey revealed that students regarded the practice exercises as valuable for developing their programming skills. In the survey, students reported positive self-efficacy in their abilities to write programs and solve programming problems. The survey also showed that students were more confident in their ability to write and troubleshoot programs than in their ability to identify program requirements. Students' actual efficacy to complete programming and problem-solving exercises can be observed from the percentage of fully completed practice exercises that they achieved throughout the course. As the semester progressed, students produced lesser error-free solutions on average, possibly due to the complexity of activities requiring students to combine more programming concepts. Future studies could investigate how students calibrate their perceived self-efficacy at various points in the semester. These studies could also explore how assignment grades and feedback provided by the instructor influence their perceptions of efficacy.

Feedback on the correctness of the assignment solutions submitted by students was an essential part of the instructional design discussed in this paper. Regular, structured feedback for each programming problem provided a way for students to focus their efforts on the different steps of the programming process. However, the feedback structure did not provide students with insight into their progress over the semester. For example, students did not receive explicit feedback on the improvement in their error-correcting skills at a given time compared to their skills at the beginning of the semester. Future implementations of instructional design could devise a method to show students how much they have progressed in effectively completing various stages of the programming process depicted in Figure 2. Additionally, the feedback mechanism could highlight the number of concepts students have mastered by

completing practice exercises throughout the course.

8. ACKNOWLEDGEMENTS

The author would like to thank and acknowledge the PASSHE-FPDC grant for funding the instructor during the summer of 2019. The grant funding helped the author acquire the required professional development in the SRL and learning design study applied in the current research.

9. REFERENCES

Bandura, A. (1996). Social Cognitive Theory of Human Development. In T. Husen & T. N. Postlethwaite (Eds.), *International Encyclopedia of Education* (2nd ed., pp. 5513-5518). Oxford: Pergamon Press.

Bandura, A. (2012). *Self-Efficacy: The Exercise of Control* (13th ed.). New York, NY: Freeman.

Beaubouef, T., & Mason, J. (2005). Why the High Attrition Rate for Computer Science Students. *ACM SIGCSE Bulletin*, 37(2), 103-106. https://doi.org/10.1145/1083431.1083474

Cleary, T. J, & Zimmerman, B. J. (2012). A Cyclical Self-Regulatory Account of Student Engagement: Theoretical Foundations and Applications. In S. L. Christenson & W. Reschley (Eds.), *Handbook of Research on Student Engagement* (pp. 237-257). New York, NY: Springer Science.

Cleary, T. J., Zimmerman, B. J., & Keating, T. (2006). Training Physical Education Students to Self-Regulate During Basketball Free Throw Practice. *Research Quarterly for Exercise and Sport*, 77(2), 251-262. https://doi.org/10.1080/02701367.2006.10599358

Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. In L. B. Resnick (Ed.), *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser* (pp. 453-494). Lawrence Erlbaum Associates, Inc.

Cooper, G., & Sweller, J. (1987). Effects of Schema Acquisition and Rule Automation on Mathematical Problem-Solving Transfer. *Journal of Educational Psychology*, 79(4), 347-362. https://doi.org/10.1037/0022-0663.79.4.347

Dunning, D. (2011). The Dunning-Kruger Effect: On Being Ignorant of One's Own Ignorance. In J. M. Olson & M. P. Zanna (Eds.), *Advances in Experimental Social Psychology* (Vol. 44, pp. 247-296). Academia Press.

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., & Mehandjiev, N. (2004). Meta-Design. *Communications of the ACM*, 47(9), 33-37. https://doi.org/10.1145/1015864.1015884

Huang, X. (2016). Example-Based Learning: Effects of Different Types of Examples on Student Performance, Cognitive Load and Self-Efficacy in a Statistical Learning Task. *Interactive Learning Environments*, 25(3), 283-294. https://doi.org/10.1080/10494820.2015.1121154

Kinnunen, P., & Malmi, L. (2006). *Why Students Drop Out CS1 Course?* ACM Press. https://doi.org/10.1145/1151588.1151604

Laurillard, D. (2012). *Teaching as a Design Science: Building Pedagogical Patterns for Learning and Technology* (1st ed., pp. 82-83). Routledge.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). Programming, Problem Solving, and Self-Awareness. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/2858036.2858252

Moores, T. T., & Chang, J. C. J. (2009). Self-Efficacy, Overconfidence, and the Negative Effect on Subsequent Performance: A Field Study. *Information & Management*, 46(2), 69-76. https://doi.org/10.1016/j.im.2008.11.006

Paas, F. G. W. C. (1992). Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach. *Journal of Educational Psychology*, 84(4), 429–434. https://doi.org/10.1037/0022-0663.84.4.429

Pajares, F., & Miller, M. D. (1994). Role of Self-Efficacy and Self-Concept Beliefs in Mathematical Problem Solving: A Path Analysis. *Journal of Educational Psychology*, 86(2), 193-203. https://doi.org/10.1037/0022-0663.86.2.193

Renkl, A. (2011). Instruction Based on Examples. In R. E. Mayer & P. A. Alexander (Eds.), *Handbook of Research on Learning and Instruction*. (pp. 272-295). Routledge.

Renkl, A. (2014). The Worked-Example Principle in Multimedia Learning. In R. E. Mayer (Ed.), *The Cambridge Handbook of Multimedia Learning* (2nd ed., pp. 391-412). Cambridge University Press.

van Gog, T., Kester, L., & Paas, F. (2011). Effects of Worked Examples, Example-Problem, and Problem-Example Pairs on Novices' Learning. *Contemporary Educational Psychology*, 36(3), 212-218. https://doi.org/10.1016/j.cedpsych.2010.10.004

van Gog, T., & Rummel, N. (2010). Example-Based Learning: Integrating Cognitive and Social-Cognitive Research Perspectives. *Educational Psychology Review*, 22(2), 155-174. https://doi.org/10.1007/s10648-010-9134-7

Weintrop, D., & Wilensky, U. (2017). Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education*, 18(1), 1-25. https://doi.org/10.1145/3089799

Zimmerman, B. J. (2009). Self-Regulation: Where Metacognition and Motivation Intersect. In A. R. Moylan (Ed.), *Handbook of Metacognition in Education* (pp. 311-328). Routledge.

AUTHOR BIOGRAPHY

Pratibha Menon is an associate professor at the Department of Computer Science and Information Systems, Pennsylvania Western University. Dr. Menon has over ten years of experience teaching courses in Computer Information Systems and has actively participated in teaching and learning innovation projects. In addition, Dr. Menon has been the Principal Investigator for grant projects that have resulted in the study of learning design for introductory courses in information systems.

APPENDICES

Appendix A. Course Objectives and Topics.

A. Objectives of the Course:

Upon completion of this course the student will be able to do the following items using the presently adopted language for this course (Fall 2010: Java):

a) Analyze business case studies and discuss strengths and weaknesses of various potential solutions.
b) Recognize and use problem-solving techniques and methods of abstract logical thinking to develop and implement structured solutions to given software design problems.
c) Apply problem-solving techniques and design solutions to business problems and implement these solutions by writing computer programs.
d) Write well-structured business programs.
e) Evaluate and debug programs.
f) Work in collaborative groups.

B. Catalog Description:

This course provides students with an understanding of business problems that are typically solved by writing computer programs, problem-solving techniques to enable students to design solutions and programming skills learned in a traditional CS1 course. Emphasis is placed on efficient software development for business-related problems. Students are required to write, test, and run programs. Prerequisite: High School Algebra or Equivalent. Three credits.

C. Outline of the Course:

a) Problem Solving Techniques for Business Problems
   i) Business Case Studies
   ii) Problem Identification and Understanding
   iii) Solution Planning (flowcharts, pseudocode, etc.)
   iv) Algorithm Development
b) Programming Concepts
   i) Structure of a Program (“Hello World”)
   ii) Constants, variables, and data types
   iii) Arithmetic operators
   iv) Relational operators
   v) Logical operators
   vi) Assignment statements
   vii) Input and output
   viii) Selection (if/else and switch)
   ix) Repetition (while, do/while, and for)
c) Strings
d) File Processing
e) Functions (in presently adopted language, “method’’)

Appendix B. Practice Exercises Examples

1. A Sample DIY Problem (to be solved in about an hour per problem):

Shopping Cart – Create a file called ShoppingCart.java
Please refer to the code demo called **VariableDataEntry.java** before attempting this problem. This problem shows you how to:

- obtain data from the user, scan this data and save it in an appropriate variable.
- perform arithmetic using the numeric data types,
- print a message displaying values of all the variables.

In this program, you will capture data of an item for a ShoppingCart application. Your program may need to know the following properties: customer_name, item_name, item price, sales tax rate, item quantity, calculated total price of all items in the cart
A ShoppingCart may need the following behaviors:

- Obtain the following data from the user for a single item: customer_name, item_price, sales_tax_rate, item_quantity. Scan these values and store them in variables of appropriate data type.
- Calculate the total price of all items in the cart
- Print a message listing all the item variables with their total calculated price (that includes the sales_tax factored in).

2. A Sample Hack-the-Code Activity (to be solved in about 15 minutes per problem):

Refer to the code called AgeCheckerCase2.java.

```
import java.util.Scanner;
public class AgeCheckerCase2 {
    public static void main(String[] args){

        //Declare the input variable
        int age = 0;

        //Declare output variable
        double ticketPrice = 0.0;

        //Declare other variables
        double salesTax = 0.05;

        //Decision structure
        if(age < 12) {
            salesTax = 0.2;
        }

        ticketPrice = ticketPrice*(1+salesTax);

    }
}
```

Hack this code so that your decision structure calculates the ticket price based on the following rule: For an age that is less than 12, give a 20% discount on ticket price, but for age greater than 65, give just a 10% discount on the ticket price for all other age groups between and including 12 and 65, give just 2% discount on ticket price.

3. A Sample Test-Tube Activity (to be solved in about 15 minutes per problem)

```
public class TracingWhileLoops_3 {
    public static void main(String[] args){

        int i = 0;
        int result = 0;
        int gate = 5;
        int n = 1;

        while(i<gate){

            if(i/4 == 0){
                result = result + 1;
            }
         i = i + n;
        }
    }
}
```

1) Determine the value of the result, i/4 and (i<gate) for each iteration of the while loop and complete the table shown below

gate = 5	n =2	i	result	i/4	i<gate
5	2	0	0		
5	2				
5	2				
5	2				
5	2				
5	2				

2) Determine the value of result, i/4 and (i<gate) for each iteration of the while loop and complete the table shown below for a gate = 10 and n = 3. Add more rows if needed.

gate = 10	n =3	i	result	i/4	i<gate
5	2	2	0		
5	2				
5	2				
5	2				
5	2				
5	2				

4. A Sample Messed-up Code Activity (to be solved in about 15 minutes per problem)

Problem: Use decision structures to check if a variable **userLetter** is a vowel in the English alphabet. Assume the value of userLetter is already obtained from the user and set to an appropriate data type in each of the following responses. Correct the errors in each of the following responses that assume a given data type for userLetter,

Response 1: userLetter is a String.

```
if (userLetter.equalsIgnoreCase "a"){ System.out.println("Letter is a vowel"); }
else if (userLetter.equalsIgnoreCase "e"){ System.out.println("Letter is a vowel"); }
else if (userLetter.equalsIgnoreCase "i"){ System.out.println("Letter is a vowel"); }
else if (userLetter.equalsIgnoreCase "o"){ System.out.println("Letter is a vowel"); }
else if (userLetter.equalsIgnoreCase "u"){ System.out.println("Letter is a vowel"); }
else { System.out.println("Letter is not a vowel"); }
```

Response 2: userLetter is a char

```
If (user == a){ System.out.println("It's a vowel");}
else if (user == e){ System.out.println("It's a vowel"); }
else if (user == i){ System.out.println("It's a vowel"); }
else if (user == o){ System.out.println("It's a vowel"); }
else if (user == u){ System.out.println("It's a vowel"); }
else { System.out.println("Not a vowel"); }
```

Response 3: userLetter is a String and you need to use a || in your if condition

```
if (letter.equalsIgnoreCase("A||E||I||O||U")){
 System.out.println("You got a vowel");
}
```

Appendix C. A Sample Set of Practice Activities and the Skills They Help Develop

List of Practice Activities used in the assignments and the problem solving skills/strategies they target							
		Activity ./ Program Name	Develop Sequence & logic	Testing	Alter the code	Trace variables	Fix errors
Module 1- Variable, Data types, Scanner methods	HTC	Age					
	MC	CharDemo					
	HTC	StringDemo					
	TT	Mice - 10 problems					
	TT	PeopleKnown					
	DIY	ShippingCost					
		TacoPrice					
		TypeCasting					
		FlooringCost					
		HealthData					
		MakeChange					
Module 2 - Decision Structures	HTC	FindSpecialValues					
	TT	RangeChecker					
	TT, HTC	AgeChecker2-5 problems					
	TT, HTC	AgeChecker3 -5 problems					
	TT, HTC	ScoreDifference-5 problems					
	TT, HTC	AgeChecker4 - 5 problems					
	TT						
	DIY	AgeZipCodechecker					
		YearToCenturyConverter					
		TicketPrice					
		AgeChecker1					
		Electrice Power Consumption					
		RockPaperScissor					
		LaborCharge					
Module 3 - Loops	TT, HTC	LoopSimulators -4 problems					
	TT	Tracing While loops -3					
	TT,HTC	While loop counters1					
	TT,HTC	While loop counters2					
	TT,HTC	While loop counters3					
	HTC	InterestCaluclator					
	HTC	DivideByTwo					
	DIY	ABCounter ForLoop					
		ABCounterWhileLoop					
		WhileLoop_SentinelValue					
		FutureTuition					
		InsectGrowth					
		ValidatingUsers					
		ValidatingUsers- part2					
		TaxProblem					
Module 4- Methods	HTC	FacePrinter_1					
	MC	PrintShapes					
	TT, HTC	FacePrinter_parameters					
	TT,HTC	DinnerPriceCalc_1					
	TT,HTC	Price Calc- 4 methods					
	DIY	DinnerPriceCalc_2					
		eBayFee					
		LengthConvertors					
		TicketingApplication 1					
		TicketingApplication 2					
		TaxApplication					

DIY - graded activities-to develop a programing solution for a given problem

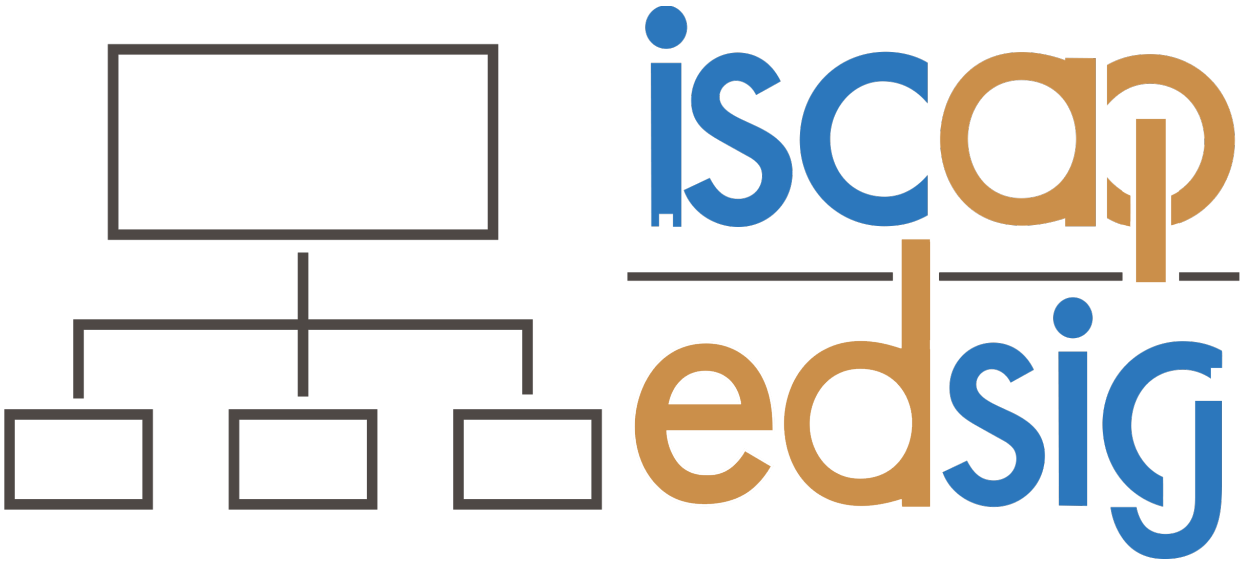
Practice Activities on pre-written code -graded

Ungraded- but students could use these skills

Appendix D. Sample Rubric Used to Assess and Provide Feedback on the DIY Assignment Problems for Each of the Four Modules

	Student is able to write the order of statements correctly, as required to meet the requirements of the problem	Student is able to identify the correct type of statements required to solve the problem	Student is able to identify the correct type of expressions to compose the statements	Student is able to write all the expressions correctly	Student is able to correctly identify the variables and its data types required to capture the data in the problem	Student is able to obtain the required inputs, as required by the problem	Student is able to correctly output data as per the problem requirements
Assignment 1:							
Assignment 2: Statements with expressions, input and output							
Assignment 3: Statements with variables,expressions, input and output							
Assignment 4: Statements with if.else / switch , variables, expressions, input and output							
Assignment 5: Statements with various types of loops,variables, expressions, inputs and outputs							
Assignment 6: Statements with if..else, loops, variables, expressions, inputs and outputs							

Note: Assignments 4, 5, and 6 cover contents of Modules 2, 3, and 4, respectively.



Information Systems & Computing Academic Professionals

Education Special Interest Group

STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the ***Journal of Information Systems Education*** have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN: 2574-3872 (Online) 1055-3096 (Print)