Education and Information Technologies
https://doi.org/10.1007/s10639-022-11247-6

Engaging students in active exploration of programming worked examples

Sebastian Garces[1] · **Camilo Vieira**[2] · **Guity Ravai**[1] · **Alejandra J. Magana**[3]

Received: 22 April 2022 / Accepted: 19 July 2022

Abstract
Worked examples can help novice learners develop early schemata from an expert's solution to a problem. Nonetheless, the worked examples themselves are no guarantee that students will explore these experts' solutions effectively. This study explores two different approaches to supporting engineering technology students' learning in an undergraduate introductory programming course: debugging and in-code commenting worked examples. In a Fall semester, students self-explained worked examples using in-code comments (n = 120), while in a Spring semester, students debugged worked examples (spring n = 101). Performance data included the midterm and final exams. Prior exposure to programming courses was taken from a survey at the beginning of each semester. Findings suggest that both the debugging and explaining forms of engagement with worked examples helped students with no prior programming experience to succeed in the course. For the worked examples to be effective, those need to be provided with some explicit form of engagement (i.e., debugging or self-explaining). Combining both strategies following explaining first and debugging second may result in a more effective approach.

Keywords Programming · Novice · Learning · Strategies · Commenting · Debugging · Worked examples · Schemata · Models · Cognitive load

1 Introduction

Jobs in computer science and Information Technology (I.T.) related fields represent a growing sector. According to the Southern Regional Education Board (SREB), by 2020, 4.6 million out of 9.2 million jobs in STEM fields (Science, Technology, Engineering, and Math) in the U.S. will be computer-related, and 3.8 million jobs will be in computer science ([SREB], 2016; Kaczmarczyk & Dopplick, 2014). Furthermore,

Extended author information available on the last page of the article

Published online: 02 September 2022

The U.S. Bureau of Labor Statistics ([BLS], 2015) estimates that by 2024 nearly 4.6 million high-wage jobs will be in computing and related fields. However, most countries are not on track to meet the labor market demand in computing fields. Meeting the demand in these innovation-intensive fields requires the governments to greatly expand and diversify their computing workforce ([SREB], 2016). Several efforts have been conducted to increase the coverage of Computer Science in K-12 education, but these kinds of policies are only starting to be implemented country-wide in some specific nations. There are some initial efforts to increase the number of hours that students are exposed to computing ([SREB], 2016), but such efforts are still lacking concrete ways to provide a thriving environment to succeed at the college level within computing coursework in engineering programs. Hence, engineering students may not be well prepared to do so – they may not have the computational or mathematical thinking preparation to succeed in college-level coursework (desJardins, 2015).

One of the challenges to fostering a proper learning environment for programming skills at the college level is the inability to address students' differences in prior exposure to computing skills. Therefore, a good approach is to create a combination of instruction by shifting from teaching to learning: "as one moves along the behaviorist—cognitivist— constructivist continuum, the focus of instruction shifts from teaching to learning, from the passive transfer of facts and routines to the active application of ideas to problems" (Ertmer & Newby, 2013, p. 58). Addressing this challenge is particularly important for three reasons: (1) students come from different backgrounds with different levels of the skills required to succeed in this environment ([SREB], 2016); (2) programming skills require high levels of cognitive processing; and (3) the development of the required mental structures is not easily attainable by students with low exposure to the topics (Robins et al., 2003). Therefore, identifying effective strategies that address the diversity in student experiences and the cognitive challenges of learning computer programming is crucial. In this context, existing strategies that get the students actively involved in their learning process include debugging and self-explaining. Debugging is a common practice in programming, a skill that is difficult to teach and learn (McCauley et al., 2008); but it is also a learning activity where students need to actively monitor their understanding of the code to identify and fix a bug. Self-explaining is an important strategy to engage students in an active exploration of worked examples instead of just superficially reading them. By self-explaining, students make sense of the learning material using what they already know. This study aims to identify the effect of scaffolding methods on supporting novice learners to attain learning outcomes in introductory programming. Specifically, this study aims to identify the effect of two chosen scaffolding methods, one being worked examples, and the other being debugging, on student academic performance. The research questions that guide this study are:

RQ1. What is the effect of engaging students with previous heterogeneous programming experiences in self-explaining or debugging on their learning of programming skills?

RQ2. What are the effects of worked examples with different engagement strategies, such as debugging examples or commenting examples, on students' programming performance within an introductory programming course?

2 Challenges to learning computer programming

As the demand for engineers with computing skills and students' interest in coding have increased in recent years, introductory programming courses have become more popular (Ullah et al., 2018). However learning to program is difficult (Magana et al., 2017; Mselle & Twaakyondo, 2012), and introductory programming courses have been demonstrated to be quite challenging for novice students (Robins et al., 2003; Ullah et al., 2018). Moreover, since there are no standard K-12 CS curricula, students arrive with different experiences and expertise in the college-level computing courses in engineering programs. Novice programmers may be overwhelmed by all the interacting elements they need to learn at once (Sweller, 2011). They may try to understand the programming code line by line, while experts are able to identify an overall abstract explanation of the program (Whalley & Lister, 2009). On the other hand, expert programmers (a) have specialized schemata; (b) organize their knowledge according to functional characteristics such as the nature of the underlying algorithm; (c) use general problem solving strategies and specialized strategies when required; (d) efficiently decompose and understand programs; and (e) are flexible in their approach to program comprehension (Von Mayrhauser & Vans, 1995). Thus, the more experienced students may begin programming courses with some of the aforementioned schemata.

Another challenge comprises the difference between declarative knowledge (e.g., being able to state how a loop works) and programming strategies (the way knowledge is used and applied, e.g., using a loop appropriately in a program) (Davies, 1993; Robins et al., 2003). There is a critical difference between programming knowledge and programming strategies. When most of the introductory programming courses focus on specific programming knowledge, which is still essential to learning a programming language, there should be an approach toward developing programming strategies (Robins et al., 2003). There is considerable evidence suggesting that programming should be approached as cognitive chunks, putting together pieces of knowledge together to get an overall better idea of the purpose of a program (Rist, 1995).

Common challenges in learning computer programming have primarily been attributed to the improper management of cognitive loads. The Cognitive Load Theory describes how we process the information given a cognitive architecture and how an instructional design may support or hinder student learning (Sweller et al., 2019). The cognitive architecture comprises a working memory and a long-term memory. Our working memory is limited in time and space, while long-term memory is vast. The working memory is loaded by (a) the intrinsic load (i.e., the required information that needs to be learned), (b) the germane load (i.e., also called germane resources, represent what we use to connect what we already know in our long-term memory

to make sense out of novel learning materials), and (c) the extraneous load (i.e., unnecessary information in the learning materials). The extraneous load should be minimized to prevent students from getting overwhelmed when processing relevant and irrelevant information together.

When learning computer programming, the learners need to understand many different things simultaneously, including the goal of the problem, algorithm design strategies to achieve that goal from the initial state, the programming language syntax, and semantics, how the computer works, and in some languages such as C, how to manage memory in the computer. Therefore, instructional materials should not include irrelevant information (i.e., extraneous loads) and maximize the connection to existing knowledge in the long-term memory (i.e., germane resources). For example, block-based programming languages have emerged as effective languages for novice learners as they remove syntax errors (i.e., irrelevant for novice learners) and provide immediate feedback without complex lines of code (e.g., languages such as Java require complex instructions for implementing a user input or a graphical user interface). Instructional design approaches that focus on problem-solving without the required scaffolding may also be overwhelming to novice programming learners. These learners often lack the required schemata to manage the cognitive loads when engaged in means-ends analysis for problem solving (Sweller, 1988). They often start from the initial problem state (i.e., the "givens") and try to feel the gap to the goal of the problem. Thus, several studies have used the Cognitive Load Theory (Sweller et al., 1998) to inform the design of learning environments that support and investigate student management of cognitive loads (Dahlen et al., 2020; Mason et al., 2016; Vieira et al., 2019). The next section describes some of these approaches to support student learning.

3 Engaging students in learning computer programming skills

Several scaffolding methods have been implemented to support novice learning of computer programming. While pedagogical strategies are teacher-centered approaches to orchestrate or deliver learning (e.g., Sentance et al., 2019; Xie et al., 2019), scaffolding methods are more student-centered approaches for supporting independent learning (e.g., Mbogo et al., 2013; Restrepo-Calle et al., 2019). For instance, effective forms of pedagogical strategies include block-based programming languages to reduce the syntax errors and provide immediate feedback (Weintrop & Wilensky, 2017), pair programming (Campe et al., 2020; Lewis, 2011), and peer instruction (Porter & Simon, 2019). Similarly, effective forms of scaffolding methods include providing worked examples with self-explanation (Vieira et al., 2017), subgoal labeling activities (Morrison et al., 2015), and automatic assessment systems for programming education (Restrepo-Calle et al., 2019; Ullah et al., 2018).

Worked examples have been studied in several fields as an approach to develop early schemata on novice learners to then engage them in problem solving (Sweller et al., 2019). A worked example is an expert's solution to a problem, comprising a problem statement, a step-by-step solution, and auxiliary representations of the phenomenon (Atkinson et al., 2000). Novice learners who engage in the study of a

worked example start identifying the required steps and strategies to solve a problem, which help them manage cognitive loads in future problem-solving activities. In order to be effective, the design of worked examples should consider a set of instructional principles that have demonstrated to be effective (Atkinson et al., 2000). The intra-example features include highlighting the steps or subgoals that need to be taken to arrive at the solution of the problem so students can easily identify them. At the same time, students need to be guided to focus on one step at a time so they do not need to focus on two different sources of information simultaneously during the learning process (i.e., the split attention effect) (Renkl, 2005). The inter-example features include providing more than one example with similar problem statements but structural differences so students can compare and contrast the solutions to identify the key features of the solutions that need to be present. This principle also suggests pairing examples and practice problems so that students engage in problem solving using what they learn from the examples. Finally, the environmental settings also represent an important instructional principle for the implementation of worked examples. Using self-explanations or explaining-to-each-other activities may engage students in a constructive exploration of worked examples (Chi, 2009), enhancing the connection between students prior knowledge and the example, filling the gaps of the learning materials, and promoting student metacognitive skills (Chiu & Chi, 2014).

Recent studies have explored how to implement these principles for computer programming (Vieira et al., 2015), and some have suggested a use-modify-create approach to integrating worked examples (Lee et al., 2011; Lytle et al., 2019). The learners first use a worked example, engaging in activities such as explaining, variable tracing, or outcome prediction (Lopez et al., 2008). Self-explanation activities may be integrated into the classroom settings using approaches such as in-code comments (Vieira et al., 2017) or subgoal labeling (Decker et al., 2019). For the next step, students engage in modifying the example. This approach is similar to providing incomplete or incorrect worked examples, or fading the worked examples, so the learners have some scaffolding with a partial solution but need to engage in problem solving themselves. Only when the learners have developed the required schemata may they engage in an iterative create, test, and refine processes to provide a complete solution to a problem from scratch. When comparing the use-modify-create approach to a create-only approach, teachers reported that students in the create-only condition needed additional scaffolding. The use-modify-create condition helped students to develop an appropriate understanding of the program (Lytle et al., 2019).

The examples provided to scaffold student learning do not need to be correct and complete. Once students start developing a basic understanding of the concepts and practices, they may prefer to engage in problem-solving. The fading effect suggests that increasingly removing steps from the correct solution so that students complete them may support student learning (Sweller et al., 2019). An alternative approach is to introduce specific bugs within the example, so the students need to debug the program to identify and fix these bugs (Atkinson et al., 2000). Experts and novices differ in their approaches to debugging computer programs (McCauley et al., 2008). Experts spend more time understanding the program than novices do and use the variable names to guide their understanding. Experts can understand the programs as chunks of code rather than individual lines of code (Lister, 2011), which allow them

to design more effective debugging strategies. This may also be related to students also using different strategies for debugging their own code compared to debugging someone else's code (Katz & Anderson, 1987). They already have a mental model of the program for their own code, so they may use backward reasoning (i.e., starting from the output). When debugging someone else's code, students often use forward reasoning (i.e., starting to read line by line and considering the inputs rather than the outputs). Experts may be devoting more time to understand the program so that they can use backward reasoning for debugging.

4 Theoretical framework: complex learning and cognitive load theory

Computer programming is a form of complex learning (Sweller, 1988). The complexity lies on the high level of interactivity among the different elements that the learner needs to consider simultaneously (Sweller, 2011). For instance, learners need to understand the problem, algorithm design strategies, the programming language syntax, and semantics, and sometimes, even how the computer works and manages memory. These elements interact with each other and increase the complexity of the learning process. In order to understand how this interactivity affects learning and how to effectively support students to overcome this complexity, we can refer to the Cognitive Load Theory (CLT) (Sweller, 1988; Vieira et al., 2017).

As described in Sect. 2, the CLT describes a cognitive architecture that explains how we process information and what cognitive loads are involved in this process. Its basic premise is that human cognitive processing is heavily constrained by our limited working memory, which can only process a limited number of information elements at a time (Sweller et al., 2019). When learning occurs, the information in the working memory is transformed into schemata in the long-term memory, which is vast (Sweller et al., 1998). These schemata can be retrieved from the long-term memory as needed by the learner to make sense of new information. The use of these schemata allows learners to solve problems using approaches they already know to be effective. Henceforth, the importance of schemata relies on the ability to recognize problem types and the actions to take for approaching each particular situation, reducing the extraneous cognitive load (Sweller, 2011).

According to the CLT (Sweller et al., 1998), while executing a programming task, the student is exposed to different types of cognitive loads: intrinsic, extraneous, and germane loads (Vieira et al., 2017). The intrinsic load refers to the inherent complexity of the task at hand (Sweller, 2011). The only way to reduce this load is by reducing the task difficulty or the complexity of the concepts encompassed within the task. The extraneous load is not beneficial to learning and depends on how the information is presented to the student. This load can be impacted directly by the instructional approach (Sweller, 2011). The germane load refers to the load required to retrieve schemata in order to make sense of the intrinsic load (Sweller et al., 2019). Learning occurs when all the interacting elements (i.e., all the different components of the learning task that need to be considered simultaneously) have been processed as a schema in the long-term memory (Sweller, 2011). If these elements can be learned

independently from each other, their interactivity is low, as well as the complexity of the learning task. As described above, computer programming requires novice learners to consider many interacting elements simultaneously (e.g., the purpose of the program, algorithm design, programming language syntax). Hence, it is important to support students on managing this interactivity to avoid cognitive overload. Integrating worked examples into the learning environments has been identified as a strategy that can help learners manage cognitive loads (Vieira et al., 2017). A worked example allows novice learners to start developing the required schemata before engaging in problem solving themselves (Sweller et al., 2019). The development of schemata through worked examples is achieved by focusing the learners' attention on problem states and associated operators (i.e., Solutions steps), enabling them to induce generalized solutions (Sweller et al., 2019).

A potential limitation of the use of worked examples as an instructional strategy is that students may not necessarily engage in carefully studying them (Sweller et al., 2019). The active exploration of the examples is crucial because "the use of worked examples is not equally effective for a learner who just reads them versus one who actually engages in reflecting and understanding them" (Vieira et al., 2017, p. 5). Different strategies have been suggested to ensure students actively explore worked examples (Atkinson et al., 2000). For instance, the use of completion problems (i.e., partially worked examples) or incorrect examples have been suggested to effectively engage students in the field of introductory computer programming (Van Merrienboer & Krammer, 1987). Such tasks provide a given state, a goal state, and a partial solution that must be completed. Thus, these tasks may elicit students to carefully analyze the example and provide a complete solution for it (Sweller et al., 2019). In programming, this process involves debugging the code to find the bug and fix it.

Alternatively, the use of self-explanations has been described as a constructive learning activity that leads to a better understanding of an example as compared to passive or active overt activities (Chi, 2009). In the context of introductory computer programming, self-explanation activities have been explored by asking students to write in-code comments to explain a sample program (Vieira et al., 2015). In this study, we evaluate two approaches to engage students in the active exploration of the examples: (1) incorrect examples where students should engage in debugging activities; (2) written self-explanations in the form of in-code comments. By using self-explaining or debugging strategies to engage students with the worked examples, we aim to promote a student-centered approach to learning using scaffolding methods. We hypothesize that this approach may help the course instructors to manage the heterogeneity of programming experiences students arrive to their courses with.

5 Methods

This study used a quasi-experimental design to investigate the effect of two different forms of engaging with worked examples, herein called Debugging or Explaining, on student performance in an introductory programming course. The quasi-experimental design is a common approach in educational research when it takes place in authentic settings, as it is often unfeasible to randomly assign individuals to different condi-

tions. Instead, the two groups are already created, each assigned to a given condition. Performing research studies in classroom settings using quasi-experimental designs can provide learners with genuine interventions that are seamlessly deployed with minimal disturbance (Sato & Loewen, 2019). As a result, such interventions allow the observation of the effects of different forms of instructional interventions without introducing potential confounding variables and unfamiliar data collection methods (Sato & Loewen, 2019). In our context, the students enrolled in the course during the spring semester of 2018 were assigned to the debugging condition, while students enrolled in the course during the fall semester of 2018 were assigned to the self-explaining condition.

5.1 Participants and procedures

The research study took place in a first-year introductory programming course called "Introduction to C programming" on two consecutive academic semesters. The emphasis of this course is on structured programming principles and understanding the basic concepts that apply to engineering problems. The topics covered in this course are problem solving using top-down design, flowcharts to explain the program logic, selection structure, repetition structure, bitwise operations, arrays, pointers, strings, passing arguments, and sequential files. The course is typically offered in Summer, Fall, and Spring semesters for engineering technology students enrolled in electrical engineering technology, mechanical engineering technology, manufacturing engineering technology, or industrial engineering technology majors. In the spring semester of 2018, 101 students enrolled in this course, while 120 students enrolled for the fall semester in the same year. The instructor (co-author #4) was the same for both semesters. In any typical semester, the female population is about 10% of the entire class.

The course design followed a three-part structure every week: A lecture, a computer lab session, and a homework assignment. These three parts are aimed at addressing specific barriers in the programming learning process. First, given the nature of the C programming language, the instructional design was situated in the procedural paradigm. This approach was selected in order to help learners (a) not to deal with the overhead added by the use of the object-oriented programming approach and (b) generate schemata for the procedural nature of problems reducing the intrinsic cognitive load (Robins et al., 2003; Sweller et al., 2011). The lecture sections focused on the programming knowledge (i.e., How the different code structures work), while the laboratory sessions aimed to teach students programming strategies (i.e., the way knowledge is applied). Finally, the homework assignments focused on individual practice. The goal of the individual practice was to guide students in the retrieval of initially developed schemata during the first two parts of the week. Homework was a mechanism to reinforce the use of these schemata to solve programming challenges. In other words, the homework assignments were used to promote the automation of the recently acquired schemata (Sweller et al., 2011).

For each of the homework assignments, students worked on solving programming challenges. Each assignment included an activity with a worked example. Students enrolled during the spring semester (i.e., Debugging condition) were asked to debug

Fig. 1 Sample of debugging exercise

```
/*********************************
This segment is the introductory comments:
CNIT 105
Name: your name
Email:  -----@purdue.edu
Debugging Exercise 02
Date: ##/##/##

GOAL: To swap the values stored in two variables.
This exercise includes a few Build errors and a logic error.
************************************************************/

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <conio.h>

int main()
{
        int num1
        int num2

        // Input first number
        printf("Enter a number for num1: ");
        scanf("%f", num1);


        // Input second number
        printf("Enter a number for num2: ");
        scanf("%f", num2);

        // Swap two numbers:
        num1 = num2;
        num2 = num1;

        // Print two numbers after swapping
        printf("num1 =  %f \n", num1);
        printf("num2 =  %f \n", num2);

        _getch();
        return 0;
}
```

a partially worked example (Van Merrienboer & Krammer, 1987) and submit a functioning version of the code. As shown in Fig. 1, the only instructions that students received included descriptions of the general goal of the code and the types of mistakes that they would find in the code. After identifying and fixing the bugs in the code, students wrote comments to explain what changes they made to the code and submitted the final program.

On the other hand, students enrolled in the course for the fall semester (i.e., Self-Explaining condition) were asked to write in-code comments in a functioning worked example to describe what it does (i.e., self-explanations, Vieira et al., 2017), with the goal of promoting an active exploration of the example. As shown in Fig. 2, the instructions asked students to write in-code comments with the goal of explaining the purpose of the code. The students enrolled during the spring semester completed nine debugging exercises in addition to the regular weekly homework assignment. The students enrolled during the fall semester completed 11 in-code comment exercises included in the weekly homework assignments.

Fig. 2 Sample of in-code commenting exercise

```
/*********************************************************************************
Your Full Name: ----------------------------

Add comments before each group of statements to explain the purpose. You may add
short comments after some of the program statement to explain the code.
Your comments should explain the purpose / rationale for the given code.
*********************************************************************************/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <conio.h>

int get_whole();
int check (int );

int main()
{
     int input;
     int even;

     input = get_whole();
     even = check (input);

     if (even == 1)
          printf("\n\t The input is an even number. ");
     else
          printf("\n\t The input is an odd number. ");
    _getch();
    return 0;
}

int get_whole()
{
     int n;
     printf ("\t Enter whole number > 0 : ");
     scanf ("%d", &n);
     return n;
}

int check (int num)
{
    if (num % 2 == 0)
         return 1;
    else
         return 0;
}
```

5.2 Data collection method

Table 1 summarizes the data collection methods and procedures followed in this study. To identify the effect of the two approaches for engaging with worked examples, we collected performance data that included the midterm and final exams, which were part of the course. These assessment tasks were oriented to effectively measure students' proficiency in programming. The midterm exam took place after the weekly assignment number five, and the final exam took place at the end of the semester. The exams consisted primarily of multiple-choice questions. Two or three questions asked students to write code. These two instruments were used as measurements of students' ability to transfer what they learn from the examples into problems in different contexts.

The weekly assignments consisted of code generation tasks (Robins et al., 2003), which students completed as the last part of the weekly teaching unit. These tasks were focused on guiding the student through the revision of the mental models constructed during the week within the lecture and the laboratory session. As can be seen in Fig. 3, all the guidance was provided in the form of code snippets, and the students were required to write the code for each assignment.

Students submitted the corresponding debugging/commenting activity as part of the required weekly assignments. However, the interventions (i.e., debugging and commenting) were not graded but only used to support the student learning process.

Table 1 Summary of the interventions data collection methods

Sources of Data Collection	Objective	Spring	Fall
Prior Exposure Characterization	Identify the level or prior exposure to programming courses	74 Students (41 with no experience and 33 with prior experience)	120 Students (57 with no experience and 63 with prior experience)
Debugging Exercises	Partially solved worked examples requiring active exploration and completion	9 Assignments	N/A
In-code Comments Exercises	Complete worked examples requiring active exploration and self-explanation	N/A	11 Assignments
Midterm Exam	Performance measure	100 Students	120 Students
Final Exam	Performance measure	99 Students	120 Students

Fig. 3 Description of assignment five

Program Description:
Write a C program to compute the average of 5 test score, while dropping the lowest score in the group. Write the following functions to modularize the program.

1) Function displayMyInfo()
This function requires no input and returns no output. It simply displays your full name, your email address and "CNIT105 – Functions", to the screen in a box of stars.

2) Function getScore() – takes no input parameter. It returns a test score (a whole number). This function prompts the user for a whole number between 0 and 100, and returns the number as output. This function should be called for each test score from the main() function.
3) Function calcAverage() – This function take 5 test scores as input. It computes and returns the average of 4 test scores (dropping the lowest of the 5 scores). This function should be called from the main() function passing all 5 test scores to it.

4) Function findLowest () – Takes 5 test scores as input and returns the minimum of the 5 numbers as output (Assume that the numbers are distinct). This function should be called from calcAverage(), which invokes this function to determine the lowest score.

Function main()
- Declare variables with proper data type and meaningful names.
- Invoke the function to display your info to the screen.
- Invoke the function getScore() 5 times to get a value for each test score.
- Invoke the function calcAverage() passing to it all 5 scores.
- Display the average to the screen.
-

Function Prototypes: Write the function prototypes before the main function in order to extend their scope to the entire program. Write the function definitions (the code) after the main() function.

The course instructor and co-author #4 in this study scored all the assignments in both semesters.

5.3 Data analysis method

At the beginning of the course, students were asked to report the number of previous programming courses that they took in the past. Based on this information, students were initially divided into three groups: (1) No previous experience, (2) previous exposure to one course, and (3) previous exposure to two or more courses. The size of the three groups was somewhat uneven since the number of students in the group with two or more courses was quite small in comparison with the other two. Preliminary statistical analysis showed that there was no significant difference in any of the performance measures between the groups with one course and the one with two or more courses, so they were combined into two groups: (1) No experience and (2) Prior experience (see Table 1).

We used descriptive statistics to identify the measures of central tendency and dispersion in student performance for the midterm and final exams. The course instructor (author #4) and the teaching assistant for the course graded the exams using a predetermined rubric. The rubric was validated over the course of multiple semesters.

We used a two-way ANOVA to compare students' scores for each measure between the two groups (i.e., No experience and Prior experience) and the two approaches to study the worked examples (i.e., Debugging and Explaining). The data met the assumptions for normality and homoscedasticity. When the results were statistically significant, we computed the effect size eta-square. The values for eta-square may be interpreted as follows: small effect size - between 0.01 and 0.06; medium effect size – between 0,06 and 0.14; large effect size – larger than 0.14 (MRC, 2015).

6 Results

6.1 RQ1. What is the effect of engaging students with previous heterogeneous programming experiences in self-explaining or debugging on their learning of programming skills?

Table 2 depicts the descriptive statistics for both the midterm and the final exams, grouped by prior experience (i.e., No experience and Prior experience), and the approach used to engage in the study of the worked examples (i.e., Debugging or Explaining). Students with no prior experience showed lower performance in the midterm compared to experienced students, both in the spring semester (debugging) and in the fall semester (explaining). These differences faded in the final exam, where inexperienced students showed similar performance compared to experienced students in both the spring semester (debugging) and the fall semester (explaining). The two-way analysis of variance (ANOVA) for the midterm exam was statistically significant with a small effect size for prior experience ($F(191,1)=6.39$, p-value$=0.01$, eta-square$=0.024$), but not the differences on the final exam based on prior experience ($F(189,1)=0.363$, p-value$=0.19$). A possible explanation for this result is that both the debugging and the explaining strategies to engage students in actively exploring the worked examples might have helped novice programmers catch up with the more experienced students at the end of the course. However, additional work that

Table 2 Midterm and final performance during Spring and Fall semesters

Performance Measure		Spring (Debugging)		Fall (Explaining)	
		Mean	SD	Mean	SD
Midterm	No experience	72.65	13.16	78.49	11.89
	Prior Experience	76.07	13.00	82.51	9.73
Final	No experience	78.72	13.80	78.52	11.83
	Prior Experience	84.06	14.29	78.98	12.37

uses direct measures of learning after the engaging activities is needed to isolate the potential effect of confounding variables (e.g., discussions during the lectures).

6.2 RQ2. What are the effects of worked examples with different engagement strategies, such as debugging examples or commenting examples, on students' programming performance within an introductory programming course?

To identify the different effects of the two engagement strategies (i.e., debugging vs. explaining), we conducted a two-way analysis of variance (ANOVA). The ANOVA for the midterm exam was also statistically significant with a small effect size for the engaging approach ($F(191,1) = 12.49$, p-value < 0.01, eta-squared $= 0.059$). These results suggest that students in the Explaining condition (i.e., fall semester) showed significantly higher performance in the midterm compared to students in the Debugging condition (i.e., spring semester). Since there was a significant difference in the average midterm scores, we conducted an analysis of covariance (ANCOVA) for the final exam with the midterm as a covariate to identify whether the differences between groups and engaging approaches were statistically significant. The results suggest that the differences in the average score for the final exam were statistically significant with a medium effect size for the engaging approach ($F(189,1) = 25.76$, p-value < 0.01, eta-squared $= 0.068$) when controlled for the midterm exam. The interaction between the engaging approach and prior experience was not significant when controlling for the midterm score as a covariate ($F(189,1) = 3.78$, p-value $= 0.054$).

Overall, these results suggest that both engaging methods (i.e., debugging and explaining) were useful for students with no prior experience to succeed in the course, as evidenced by the similar performance in the final exam as compared to students with prior experience in computing. The results also suggest that students in the debugging condition were able to perform as well as students in the explaining condition on the final exam, despite performing significantly lower in the midterm exam.

7 Discussion

This study explored two approaches to engage students in the active exploration of worked examples: debugging and explaining. When it comes to novice learners under the scope of Cognitive Load Theory (Sweller et al., 1998) both strategies played an important role. The use of in-code comments as a self-explanation strategy engaged students in the active exploration of the worked examples helping them: (1) reflect

on what they knew and what they did not know and (2) develop early schemata for problem solving. The debugging strategy engaged students in scaffolded problem solving. Students needed to reflect on how the partial solution was constructed or reflect on how they could fix it or complete it. For this reflection on someone else's code, novice programmers will often use forward reasoning (i.e., starting to read line by line and considering the inputs rather than the outputs) (Katz & Anderson, 1987).

The results from this study showed that both debugging and explaining activities helped students with the no-prior experience to succeed in the course. Novice learners tended to take advantage of exploring worked examples, while more advanced students preferred to engage in problem solving (Sweller, 2011). The increase in performance between the midterm and final exam accounts not only for the effective building of mental models and schemata but for the effective retrieval of these models (i.e., germane load) (Sweller et al., 2019). Recent work has explored the use of in-code comments as a self-explaining strategy (Vieira et al., 2015, 2017, 2019). Vieira and colleagues (2017) found that students with different prior experiences with programming saw different affordances for self-explaining using in-code comments. For instance, novice learners found them useful for better understanding of examples and practicing algorithm design. More advanced learners with prior experiences in other languages found these activities useful in getting used to new programming syntax. Vieira et al., (2019) also found that prior experiences influence how they approach these explanation activities. While advanced programmers wrote simple explanations and assumed the code was self-explanatory, novice learners reflect on every line of code as part of their learning process. For the analysis of the active exploration strategies used in the worked examples, the results have some clear indications in terms of supporting the novices' learning process. The use of completion exercises (i.e., debugging) (Sweller et al., 2019; Van Merrienboer & Krammer, 1987) also served as a good strategy for novice learners in terms of understanding basic introductory concepts in programming and helping them build solid foundations for the required mental models. Nonetheless, as the complexity of the task at hand increased (i.e., Intrinsic load) (Sweller, 2011), debugging exercises also increased the required cognitive processes and, in consequence, the task's difficulty.

When comparing the debugging condition with the explaining condition, it seems like it was more effective than the explaining condition. Students in the debugging condition showed a larger increase between the midterm and the final exams as compared to the explaining condition. However, students in the explaining condition already had a high performance on the midterm exam, to begin with, so they may have faced a ceiling effect. Furthermore, the midterm exam did not assess all the concepts that the final exam did.

As K-12 computing education is turning into a priority globally, the diversity of experiences that students arrive with into undergraduate courses becomes a challenge for instructors. Thus, it is important to identify effective scaffolding methods that work for novice learners while still engaging more advanced students. As discussed above, existing work has demonstrated that self-explaining in the form of written comments may provide different affordances for students with different experiences (e.g., Vieira et al., 2017). Debugging, however, had not been explored as a pedagogical approach to this goal. The main contribution of this study is the evidence that

engaging students in active explorations of worked examples using self-explanation activities or debugging activities may help instructors overcome teaching challenges associated with students' previous heterogeneous programming experiences in introductory programming courses. The implications for teaching and learning relate to the effectiveness of worked examples in the context of programming, especially for novice learners. As evidenced from this and other studies that have used worked examples in the context of learning programming, for worked examples to be effective, those need to be provided with some explicit form of engagement. Debugging worked examples or self-explaining worked examples can result in positive learning outcomes. However, findings from this study also suggest that explaining worked examples might be more useful at an earlier stage of the learning process for the purpose of schemata development, while debugging the partially worked examples might first require the development of some schemata, so then novice learners can take advantage of the examples by debugging them. Students' ability to find and fix bugs in a code depends on the strength of their conceptual understanding, although existing literature also suggests that students may benefit from instructors training them on how to debug (McCauley et al., 2008). Combining both strategies following explaining first and debugging second may result in a more effective approach.

8 Conclusions, limitations, and future work

This study explored the effect of introducing worked examples with different engagement strategies (i.e., debugging or explaining) on students' programming performance within an introductory programming course. Students with no experience in programming showed lower performance in the midterm exam for both conditions as compared to students with prior experience. However, this difference faded in the final exam, suggesting that both the debugging and the explaining conditions helped students with no experience to succeed in this course. Students in the debugging condition showed a larger increase from the midterm to the final exam compared to students in the explaining condition.

There are certain limitations to this study that are worth mentioning. First, this study focused on using the aforementioned strategies in separate semesters. We consider it pertinent to conduct a future study to analyze the effects of combining both strategies for novice learners following an experimental approach. Second, the measures of performance for this study were the midterm and the final exams. As discussed above, the midterm did not assess all the concepts that the final exam did. And we did not collect baseline data beyond students' prior experience in programming courses. And only one grader (i.e., the course instructor or the teaching assistant) was used for scoring all the performance measures. Thus, future work should compare the two conditions with equivalent pretest and posttest measures and a control group. Furthermore, including qualitative data sources may provide a further understanding of how different students engage in debugging and explaining practices and what practices are more effective in taking advantage of the worked examples.

Acknowledgements This research was supported in part by the US National Science Foundation under the award number EEC 1826099. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Data availability Data are available on request due to privacy or other restrictions.

Declarations

Conflict of interest None.

References

Bureau of Labor Statistics. Employment by major occupational group, 2014 and projected 2024. Retrieved from [BLS], & Washington (2015). D.C.: https://www.bls.gov/news.release/ecopro.t04.htm

[SREB] (2016). *Bridging the Computer Science Education Gap: Five Actions States Can Take.* Retrieved from https://www.sreb.org/publication/bridging-computer-science-education-gap

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of educational research*, *70*(2), 181–214

Campe, S., Denner, J., Green, E., & Torres, D. (2020). Pair programming in middle school: variations in interactions and behaviors. *Computer Science Education*, *30*(1), 22–46

Chi, M. T. (2009). Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in cognitive science*, *1*(1), 73–105

Chiu, J. L., & Chi, M. T. (2014). Supporting self-explanation in the classroom. *Applying science of learning in education: Infusing psychological science into the curriculum*, 91–103

Dahlen, O., Lervik, A., Aarøen, O., Cabriolu, R., Lyng, R., & van Erp, T. S. (2020). Teaching complex molecular simulation algorithms: Using self-evaluation to tailor web-based exercises at an individual level.Computer Applications in Engineering Education

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

Decker, A., Margulieux, L. E., & Morrison, B. B. (2019). *Using the SOLO Taxonomy to Understand Subgoal Labels Effect in CS1.* Paper presented at the Proceedings of the 2019 ACM Conference on International Computing Education Research

desJardins, M. (2015). The real reason U.S. students lag behind in computer science.Fortune

Ertmer, P. A., & Newby, T. J. (2013). Behaviorism, cognitivism,constructivism: Comparing critical features from an instructional design perspective. *Performance Improvement Quarterly*, *26*(2), 43–71

Kaczmarczyk, D., & Dopplick, R. (2014). Rebooting the pathway to success: Preparing students for computing workforce needs in the United States. *Renee, Rebooting the Pathway to Success: Preparing Students for Computing Workforce Needs in the United States (March 05, 2014)*

Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, *3*(4), 351–399

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., & Werner, L. (2011). Computational thinking for youth in practice. *Acm Inroads*, *2*(1), 32–37

Lewis, C. M. (2011). Is pair programming more effective than other forms of collaboration for young students? *Computer Science Education*, *21*(2), 105–134

Lister, R. (2011). *Concrete and other neo-Piagetian forms of reasoning in the novice programmer.* Paper presented at the Conferences in research and practice in information technology series

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). *Relationships between reading, tracing and writing skills in introductory programming.* Paper presented at the Proceedings of the fourth international workshop on computing education research, Sydney, Australia

Lytle, N., Cateté, V., Boulden, D., Dong, Y., Houchins, J., Milliken, A., & Barnes, T. (2019). *Use, Modify, Create: Comparing Computational Thinking Lesson Progressions for STEM Classes.* Paper presented at the Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education

Magana, A. J., Falk, M. L., Vieira, C., Reese Jr, M. J., Alabi, O., & Patinet, S. (2017). Affordances and challenges of computational tools for supporting modeling and simulation practices. *Computer Applications in Engineering Education*, *25*(3), 352–375

Mason, R., Seton, C., & Cooper, G. (2016). Applying cognitive load theory to the redesign of a conventional database systems course. *Computer Science Education*, *26*(1), 68–87

Mbogo, C., Blake, E., & Suleman, H. (2013). *A mobile scaffolding application to support novice learners of computer programming.* Paper presented at the Proceedings of the Sixth International Conference on Information and Communications Technologies and Development: Notes-Volume 2

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

Morrison, B. B., Margulieux, L. E., & Guzdial, M. (2015). *Subgoals, context, and worked examples in learning computing problem solving.* Paper presented at the Proceedings of the eleventh annual International Conference on International Computing Education Research

MRC (2015). Cognition and Brain Sciences Unit (2009). *Rules of thumb on magnitudes of effect sizes.* Retrieved from https://imaging.mrc-cbu.cam.ac.uk/statswiki/FAQ/effectSize

Mselle, L. J., & Twaakyondo, H. (2012). The impact of Memory Transfer Language (MTL) on reducing misconceptions in teaching programming to novices. *International Journal of Machine Learning and Applications*, *1*(1), 388–394. doi:https://doi.org/10.4102/ijmla.v1i1.3

Porter, L., & Simon, B. (2019). A case study of peer instruction: From University of California, San Diego to the computer science community. In S. A. R. Fincher (Ed.), *The Cambridge Handbook of Computing Education Research* (pp. 861–874). Cambridge: Cambridge University Press

Renkl, A. (2005). The worked-out-example principle in multimedia learning.The Cambridge handbook of multimedia learning,229–245

Restrepo-Calle, F., Ramírez Echeverry, J. J., & González, F. A. (2019). Continuous assessment in a computer programming course supported by a software tool. *Computer Applications in Engineering Education*, *27*(1), 80–89

Rist, R. S. (1995). Program structure and design. *Cognitive Science*, *19*(4), 507–562

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172

Sato, M., & Loewen, S. (2019). Methodological strengths, challenges, and joys of classroom-based quasi-experimental research.Doing SLA research with implications for the classroom: Reconciling methodological demands and pedagogical applicability,31–54

Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education*, *29*(2–3), 136–176

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, *12*(2), 257–285

Sweller, J. (2011). Cognitive load theory. *Psychology of learning and motivation* (55 vol., pp. 37–76). Elsevier

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive load theory*. New York: Springer.

Sweller, J., van Merriënboer, J. J., & Paas, F. (2019). Cognitive architecture and instructional design: 20 years later. Educational Psychology Review,1–32

Sweller, J., Van Merrienboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296

Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A., & Saleem, F. (2018). The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, *26*(6), 2328–2341

Van Merrienboer, J. J., & Krammer, H. P. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science*, *16*(3), 251–285

Vieira, C., Magana, A. J., Falk, M. L., & Garcia, R. E. (2017). Writing in-code comments to self-explain in computational science and engineering education. *ACM Transactions on Computing Education (TOCE)*, *17*(4), 1–21

Vieira, C., Magana, A. J., Roy, A., & Falk, M. L. (2019). Student explanations in the context of computational science and engineering education. *Cognition and Instruction*, *37*(2), 201–231

Vieira, C., Yan, J., & Magana, A. J. (2015). Exploring design characteristics of worked examples to support programming and algorithm design. *Journal of Computational Science Education*, *6*(1), 2–15

Von Mayrhauser, A., & Vans, A. M. (1995). *Program understanding: Models and experiments Advances in computers* (40 vol., pp. 1–38). Elsevier

Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1–25

Whalley, J. L., & Lister, R. (2009). *The bracelet 2009.1 (wellington) specification.* Paper presented at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*(2–3), 205–253

Authors and Affiliations

Sebastian Garces[1] · Camilo Vieira[2] · Guity Ravai[1] · Alejandra J. Magana[3]

✉ Alejandra J. Magana
admagana@purdue.edu

Sebastian Garces
sgarcesp@purdue.edu

Camilo Vieira
cvieira@uninorte.edu.co

Guity Ravai
guity@purdue.edu

[1] Department of Computer and Information Technology, Purdue University, West Lafayette, IN, USA

[2] Department of Education, Universidad del Norte, Barranquilla, Colombia

[3] Department of Computer and Information Technology, School of Engineering Education, Purdue University, 401 N. Grant Street, Knoy Hall of Technology, 47906 West Lafayette, IN, United States