Arab Journal of Administrative Sciences

2019, Vol. 26, No. 3, 413 - 443

Mohammad N. AlMarzouq

Kuwait University
Kuwait

Repetition as a Strategy to Teach Business School Students Computer Programming

Abstract

Aim of the Paper: This study investigates the effectiveness of repetition as a strategy for teaching novices from a non-computer science major computer programming.
Study Design: The study follows a quasi-experimental approach, using both OLS regression and simple slopes for data analysis.
Sample and Data: The sample consists of 72 undergraduate business students in an introductory programming course.
Results: Under the guidance of an instructor, repetitive solving of practical programming assignments had a positive and significant effect on student performance. The observed effect was stronger for students with a lower GPA.
Conclusion: Repetition can be a very effective strategy in building programming competencies, more so for students of lower GPA. The study highlights the importance of time factors in training, and the need to introduce programming very early in business school curricula.

Keywords: Information system education, Computer and information systems training, Novice programmers, Introductory programming education, Programming misconceptions.

Introduction

Can we train students or employees with no background in computer science to benefit from computer programming? The literature that is relevant to answering this research question can be classified into two main streams. First, there is the literature on computer science education devoted to the skill building of computer programmers that have no prior experience with programming languages (i.e., novices). This stream of research informs us of the main challenges and misconceptions faced by educators and students alike (e.g, Meerbaum-Salant, Armoni, & Ben-Ari, 2013; Moons & De Backer, 2013; Nuutila, TÌ, & Malmi, 2005; Striegel & Rover, 2002). It also highlights the importance of this task due to the

Submitted: 8/5/2019, revised 1: 18/8/2019, revised 2: 1/9/2019, accepted: 5/9/2019.

high failure rates in introductory programming courses, even for computer science majors (Robins, Rountree, & Rountree, 2003). One might, therefore, imagine that teaching introductory computer programming to novices in majors other than computer science - such as business school majors - might pose an even greater challenge (Forte & Guzdial, 2005; Lahtinen, Ala-Mutka, & JÌrvinen, 2005; Mow, 2008; Rist, 1991).

This research stream has also introduced and empirically tested numerous programming teaching strategies to deal with these challenges and misconceptions. These strategies include the use of natural language (Good & Howland, 2017), flipped classrooms (Hayashi, Fukamachi, & Komatsugawa, 2015), collaborative learning (Serrano-Cmara, Paredes-Velasco, Alcover, & Velazquez-Iturbide, 2014), problem-based learning (PBL) (Nuutila *et al.*, 2005), and Scratch visual programming language (Meerbaum-Salant *et al.*, 2013). While these strategies have demonstrated their effectiveness in teaching novices, we have found them to be impractical to implement in business school settings. Some strategies, such as PBL, collaborative learning, and flipped classrooms, assume some level of experience that business school students and teaching staff may lack. Scratch and the natural language approach, while promising, focus on long-term programming goals, rather than endowing students with the practical skills necessary for implementing solutions to business or decision-making problems within the time frame of a single semester. We, therefore, propose a practice-based strategy that is unlikely to challenge teaching resources, built simply on the well-known adage: ''Practice makes perfect''.

This practice-based approach is informed by the second stream of research relevant to our research questions, which comes from the field of cognitive psychology. The literature on experience building through deliberate practice informs us that skill building can be improved through practice under the guidance of an instructor (Ericsson, Krampe, & Tesch-Romer, 1993; Ericsson, Prietula, & Cokely, 2007). One of the main findings of this line of research is that a person will require no less than 10,000 hours of deliberate practice to become an expert, which cannot be covered within a single semester of instruction (Ericsson *et al.*, 1993). With our practice-based strategy, we hope to give students and trainees deliberate practice within a limited timeframe to build the necessary competence to take advantage of programming languages in solving domain specific problems in their respective fields. The literature on active control of thought (ACT) theory could shed some light on how engaging in practice will provide students with practical

experience that cannot be gained by simply attending classes, reading textbooks, or engaging in written exercises. It suggests that if a programming task is not overly complex and has deterministic outcomes, then students can be trained to improve their time and accuracy in repeating such a task (Anderson, 2007). The goal of introductory programming courses should be to train students to be competent enough to complete a programming task for problem solving, within a limited timeframe, and not necessarily become experts.

We integrate in this study the insights from these two lines of research and present the practice-based strategy for introducing novices from non-computer science backgrounds to computer programming. We identify the main conditions necessary for the success of this strategy, as well as the challenges that educators need to be aware of. One of the main contributions of this work is to bring the attention of educators to the importance of computer programming as an applied skill in problem solving even for non-computer science majors. We show, with empirical support, that a simple practice-based strategy can be effective in building the necessary competence in such majors to take advantage of programming languages in problem solving. These findings will be of particular importance to curriculum designers for non-computing majors - such as business schools - where introductory computer programming can be introduced early to most majors such that students can have enough opportunity to practice and apply computer programming skills in their respective fields. This study also highlights the important role of instructors as a necessary condition for the success of the practice-based strategy.

This paper is organized as follows. First, we start with a review of the relevant literature from novice computer programming education, as well as the literature on deliberate practice and Active Control of Thought (ACT) theory. We then introduce the theoretical background that the repetitive strategy is based on, as well as the main hypotheses of this study. This is followed by a description of the methodology used, which includes the implementation of the repetition strategy that we have used in our introductory programming course, as well as our data collection method and analysis. We then discuss the results of our work, as well as the main contributions and limitations. Finally, we offer some concluding thoughts.

Literature Review

The literature on computing education has given special attention to the training of novice programmers in introductory computer programming classes.

A novice programmer is one who is introduced for the first time to computer programming and is learning his/her first programming language (Robins, 2019). What is characteristic of a novice programmer is that the knowledge he/she acquires about programming is fragile (Lowe, 2019; Perkins & Martin, 1986). Fragile knowledge is newly acquired knowledge that the student fails to apply consistently. For example, students might acknowledge that they have comprehended the concept of variables and assignment operators when they are first introduced by the instructor. But when asked to solve a problem that requires variable declaration or assignment, the students might fail to recall these concepts, or use them incorrectly.

A key concept in computing education is that of notional machines (Du Boulay, 1986). A notional machine is an abstraction, or idealization of the computer system used to depict what happens when a program is executed. Instructors create visual depictions of notional machines as a teaching aid to simplify computing concepts. When novice students learn programming concepts, they create mental models of these notional machines that might initially be simple and erroneous (Norman, 1987). These incorrect mental models that novices may have or lack of a notional system result in various misconceptions about computer programming and are the main reason why students have fragile knowledge (Perkins, Schwartz, & Simmons, 1988). Most notable of these misconceptions is the failure to correctly understand how programs are dynamically executed in a computing environment. As a result, novices fail to distinguish between a writing code and an executing code. As the programmer gains more experience, these mental models of notional machines improve in complexity and correctness, and are a better reflection of the realities of computer programing and code execution (Sorva, 2013).

The literature also highlights the important role the instructor plays in shaping the mental models formed by novice students. This important role is highlighted by showing how the different strategies we reviewed from the computing education literature can improve the effectiveness of the instructor-student relation through two main mechanisms. The first are the strategies that require the explicit involvement of an instructor or bring the student closer to collaborating with an experienced instructor that can uncover any student held misconceptions about programming through strategies such as the flipped classroom (Hayashi *et al.*, 2015), collaborative learning (Serrano-Cmara *et al.*, 2014), or problem-based learning (PBL) (Nuutila *et al.*, 2005). The second are the strategies that visualize the notional machine to both the student and instructor such that they may

uncover misconceptions and create more robust mental models, such as the Scratch visual language (Meerbaum-Salant *et al.*, 2013), or the use of natural language to describe program instructions (Good & Howland, 2017).

Our proposed practice based approach would fall into the category of strategies that require the explicit involvement of the instructor in addressing student misconceptions about programming, as well as dealing with motivational issues as students stumble through their journey to learn to program (Sorva, 2013). This engagement with students to introduce them to concepts and monitor their practice is known as 'guided practice' (Ericsson *et al.*, 1993). Guided practice is based on two principles: the first is to improve what students already know through practice and the second is to introduce new concepts to students to push them out of their comfort zones so that they can gain new skills (Ericsson *et al.*, 2007). As a result, students' knowledge becomes less fragile as they get closer to becoming experts. However, the road to becoming an expert is not easy, according to Ericsson (2007), as a student or trainee needs to put in at least 10,000 hours of practice before the expert status is attained. The goal of introductory programming classes is not to create experts, but to set the students on their path to being experts. With enough practice, students would reach their breakthrough moment and comprehend the core concepts that would make understanding programming easier (Meyer & Land, 2006). It is this breakthrough moment that we would like students to attain with practice.

Yet, one of the biggest challenges in teaching novices is that learning about programming and engaging in its practice are not the same thing. This becomes a more pronounced problem for novices that are non-computer science majors, as they lack the motivation to learn to program given its seemingly peripheral role in their majors. In addition, students might not have received sufficient training on skills that are known to be related to success in learning to program, such as mathematical problem solving skills (Konvalina, Wileman, & Stephens, 1983). As a result, students are likely to delay such introductory courses to the end of their program or be content with obtaining poor grades (Forte & Guzdial, 2005). Even some of the students we interviewed in this study confirmed this observation by pointing out that they did not expect themselves to be working in the field of application development. Therefore, they lacked the motivation to learn and perform well in introductory programming courses and typically postponed these courses to their senior year and were content with a passing grade, since they did not perceive its relevance to their chosen major. Surprisingly, such observations seem to be the norm amongst IS undergraduate students (Woszczynski, Haddad,

& Zgambo, 2005). Such students would miss the chance to practice programming early and experience how programming could be relevant to their careers. Rather unintentionally, the latest curriculum guidelines for undergraduate degrees in IS published by the Association of Information Systems (AIS) and Association of Computing Machinery (ACM) may inadvertently reinforce this view by recommending that programming courses be taught as electives (Topi *et al.*, 2010).

Active Control of Thought (ACT) theory can shed some light on why there is a big difference between learning about programming and practicing it. The theory distinguishes between two types of knowledge: procedural knowledge and declarative knowledge. Declarative knowledge is knowing about a skill or practice, whereas procedural knowledge is knowing how and when to apply this knowledge in practice. Procedural knowledge is dependent on declarative knowledge, as it cannot be gained without the application of declarative knowledge (Anderson, 1982, 2007). The theory suggests that with practice, an individual goes through three stages of experience: declarative, procedural, and automatic (Anderson, 2015), which are distinguished by the nature of the individual's knowledge at each stage. During the declarative stage, individuals know about the skill; knowledge is acquired through observation without performing the action. For example, an individual observing a tennis match might understand all the rules and actions involved but would not be able to competently play the game on that basis alone. Individuals enter the procedural stage when they start imitating a tennis player. During this stage, declarative knowledge is converted into procedural knowledge through practice. Provided the declarative knowledge is easily accessed and retrieved by the individual, learning how to rudimentarily perform the task can be accomplished with little practice, which explains the large gains made by initial practice according to the power law (Anderson, 2015; DeKeyser, 2014).

A central concept that ACT theory is built upon is the power law of practice (Newell & Rosenbloom, 1981, 2013). This law suggests that with practice, cognitive skills with deterministic calculations are replaced by simple memory retrieval. As a result, the speed, accuracy, and recall of a task should generally improve with practice. The relationship between the time to perform a task and number of practice trials follows a power curve, also known as a Pareto distribution, where the graph initially starts with a steep decline, then smooths out to a flat long tail (See Figure 1). This means that improvement from practice trials becomes significant when the individual first starts training, hence the steep decline in the graph. Subsequently, as the skill of the individual improves, gains in speed and accuracy from practice trials diminish significantly, producing the long flat tail representing small gains.

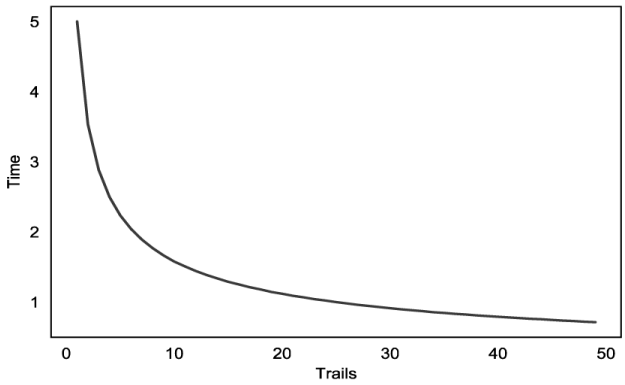

Figure 1: Power Law Distribution

Rudimentary performance of a task and becoming proficient in it, however, constitute two different things, and this is where the third stage of automaticity comes in. With diligent practice, even though there may be few observable gains, the task becomes like second nature to the individual and is performed almost automatically as the knowledge associated with it becomes less fragile. That is, the individual no longer needs to retrieve the declarative knowledge about the task and simply performs the task correctly without thinking about it or requiring a reference manual explaining how to correctly perform the task (Anderson, 1992, 2015). It is, however, important to note that achieving this automaticity is also a gradual process, which can be improved upon, albeit very slowly compared to the improvement experienced in the procedural stage. Automatic tasks might suffer from the lack of transferability to other domains because of high degrees of specificity (Anderson, 2015; DeKeyser, 2014; Singley & Anderson, 1989). Such specificity in knowledge is expected of students in introductory courses, where the goal is to apply these skills to specific problems. It is this automaticity in specific tasks that we would like novice students to achieve with a practice-based strategy in order to consider them competent users of programming languages. It is with continued practice and diligence after the introductory course that these novices will become experts that can consistently apply these skills in more general situations (Anderson, 2015; DeKeyser, 2014; Ericsson *et al.*, 2007).

The Practice-Based Strategy

Central to the success of the practice-based strategy is the role of the instructor. The instructor is expected to introduce the students to the declarative knowledge required to practice programming. This is generally the knowledge available in

textbooks and is introduced during classes. In addition, the instructor plays a pivotal role in ensuring that the success conditions for a practice-based strategy are in place. Without upholding these conditions, students would not see any additional benefit from the practice-based strategy when compared to learning directly from a textbook or an online resource.

The first of these conditions is that the instructor be aware of the importance of practice. Based on ACT theory, the practice of programming will be an entirely different knowledge-gaining experience for students from reading textbooks. What becomes evident to both students and instructors once engaged in practice, is that writing computer programs is a complex cognitive task (Robins *et al.*, 2003; Singley & Anderson, 1989). The literature on computer programming education we have reviewed focuses mostly on the coding aspect of computer programming. However, the task involves many elements that programmers need to master, including problem understanding, design, coding, and maintenance (Pennington & Grabowski, 1990). All these tasks are interrelated, and programmers might simultaneously work on multiple different subtasks, which creates a significant barrier for novices to overcome (Du Boulay, 1986). The programming task cannot be completed without completing these interrelated subtasks. How well a programmer performs these subtasks is what sets a competent programmer apart from a novice one (Winslow, 1996). Therefore, improving programming skills is tantamount to improving these tasks.

A direct result of the nature of the programming task is that an instructor should not assume it to be similar to problem solving in other domains, even though the two skills can be highly related (Konvalina *et al.*, 1983; Wilson & Shrock, 2001). For one thing, the constructs used in solving programming problems can be very different and, therefore, students must employ an entirely different mental model to solve a mathematics problem versus a programming problem. For programming, lists, dictionaries, recursion, and conditionals are used in solutions, but similar constructs are obviously not used to solve math-based problems related to finance or accounting, for example. Transferring knowledge from the domain of pure problem solving to programming would be a virtually impossible challenge for a novice programmer without the guidance of an instructor (Rogalski & Samurçay, 1990).

Another condition is that the instructor should be aware of the misconceptions that could be held by students. These misconceptions are mental obstacles that

might impede students' ability to learn, or might result in mistakes that could prevent them from correctly composing programs (Sorva, 2013). A very important misconception that became evident to us only when students started to practice programming is related to how students perceive computer programs to work (i.e., their held notional machine). The realization that programmers do not themselves solve problems, but rather instruct computers on how to solve problems, can be difficult for novices (Sorva, 2013), who, before programming any solutions must first understand how computers operate and how to express solutions in ways the computer will understand (Du Boulay, 1986). For example, how data is loaded into memory and manipulated using loops and functions are issues not considered in pure problem-solving domains. To make matters worse, developers also must be aware of the role the user plays in executing the solution. While the developer might successfully write a program that instructs the computer to solve a problem, the problem cannot be solved unless the user executes the program. Considering that some solutions require user input, and that solutions can be organized into abstract functions, understanding programming solutions and how the developer, computer, and user interact with the program can be a daunting task for novices. In the case of introductory programming classes, application development is often simplified by treating the developer and user as the same entity. While this may be a useful simplification for experienced developers, for novices who do not yet understand the interplay between computers, developers, and users (i.e., the dynamic execution environment), this kind of simplification might unintentionally lead to more confusion (Sorva, 2013). Therefore, the guidance of an instructor who understands these challenges, and their constant monitoring and feedback to students, is a necessary condition for the success of our proposed practice-based strategy.

Finally, instructors should be aware of the limits of their students and constantly monitor their progress and how well the students have comprehended new content. Care should be taken to not overwhelm the students with so much knowledge as cognitive overload can cause students to fail to learn or perform exercises properly (Fitzgerald *et al.*, 2008). Instructors should plan some time for the introduction of new declarative knowledge to students followed by enough practice time to give students the opportunity to convert this declarative knowledge into procedural, while keeping an eye for any misconceptions in student understanding.

To avoid misconceptions and cognitive overload, we suggest that any introductory programming course start with a general overview of how programs are written and how they are executed. Instructors should also be aware when first

introducing programming, that each of the programming subtasks (i.e., designing, coding, and maintenance) will also have their own specific declarative knowledge that students must acquire and convert to procedural knowledge through practice. Therefore, the first few classes will be overwhelming for the students due to cognitive overload. Because these first classes are critical, instructors are advised to take small steps until students have mastered the programming workflow. With enough practice, the programming task will be second nature to students (i.e., automatic). Once students reach this milestone, they would have passed a very important threshold in which they can make significant progress in learning to program (Meyer & Land, 2006).

For example, some Integrated Development Environments (IDEs), such as Visual Studio or Eclipse, provide students with the means to design, code, and maintain a programming solution. However, the novice user still needs to acquire the declarative knowledge specific to what the design, coding, and maintenance tasks entail, as well as the declarative knowledge specific to using the IDE to perform these tasks, before using such a tool becomes second nature. Therefore, simplicity in tool choice is important in order to reduce the students' burden of learning and practice. However, once the students grasp what design, coding, and maintenance generally entail, they will no longer have difficulty distinguishing between the idiosyncratic knowledge related to the used tools and the programming subtasks themselves. As a result, they will be in a better position to understand the function of more complex tools.

To summarize, assuming that an instructor who is aware of the challenges and misconceptions faced by novice programmers guides students, and who gradually introduces students to new concepts to avoid cognitive overload, then the students will become more competent with practice in using programming languages to solve domain specific problems. Students need not only internalize the declarative knowledge related to the programming task and code execution, but also the specific knowledge associated with the tools used to performing each of the programming subtasks of designing, coding, and debugging. The instructor will play a pivotal role constantly reminding students and pointing out gaps and misconceptions in their knowledge. However, with constant reminding of the declarative knowledge and with more practice repetitions, students will eventually recall declarative knowledge easily from memory. As a result, declarative knowledge is converted with time and practice to more consistent procedural knowledge, making students more competent in using programming languages (Anderson, 2015). Therefore:

H1: *With more practice repetitions, novice programmers become more competent in using programming languages.*

Past academic performance has long been shown to be related to the success of students in introductory programming courses (Bauer, Mehrens, & Vinsonhaler, 1968; Butcher & Muth, 1985; Fowler & Glorfeld, 1981; Karim, Carroll, & Long, 2016; Konvalina et al., 1983; Newsted, 1975). As a result, one would expect that students with higher GPAs to perform highly in introductory programming courses regardless of the teaching strategy employed by the instructor. Therefore, the success of practice-based strategy hinges on its ability to successfully train students, regardless of their prior academic performance. The importance of academic ability lies in the fact that instructors have no control over the abilities of students admitted to the introductory programming course, yet they have a direct impact on students' ability to learn. Therefore, instructors need to be aware of differences between student abilities should there be any differences in how students react to the practice-based strategy so that they may adapt accordingly. Therefore, we expect that a successful practice-based strategy would be more effective in improving the competence of students of low academic ability, than students with high academic ability, hence:

H2: *With more practice repetitions, students that generally perform poor academically would observe higher improvements in their competence in using programming languages than those with good academic performance would.*

Equally important is to determine whether gender will have an impact on student learning performance so that instructors can adjust to different student needs. Prior research suggests that males and females may differ in their attitudes toward computers (Colley & Comber, 2003; Kay, 2006), may prefer different learning styles (Lau & Yuen, 2011; Malik & Coldwell-Neilson, 2018), and may differ in their aptitude toward computer programming (Rubio, Romero-Zaliz, Ma& de Madrid, 2015; Wagner, 2016). As such, we considered whether there might be any gender-specific considerations impacting the effectiveness of repetition as a learning strategy. Should a difference between genders exist, then we will also observe a differential effect for the practice-based strategy on the improvement in competence of different gender groups. Therefore:

H3: *With more practice repetitions, male students would observe higher improvements in their competence in using programming language than females would.*

Methodology

We implemented the practice-based strategy in an undergraduate business administration course that introduces students to computer programming. Students were required to implement data-oriented desktop applications. The requirements of these applications are specific, and students can, therefore, be trained to recognize and implement the most appropriate programming solution for a limited set of problems. This strategy required careful planning on the part of the instructor concerning how topics should be introduced in a way that consistently challenged the students but never overwhelmed them. Instructors introduced the relevant material (i.e., declarative knowledge) at the beginning of each class and ensured that students had ample opportunity to apply this knowledge in small practice exercises.

In our sample course, the first week of instruction was dedicated to explaining how computers work and introducing the interplay between the developer, computer, and user. The students were reminded throughout the course about this interplay whenever they seemed to be confused about how to write their solutions. For example, when a student did not understand when to ask for input or from whom, that signaled that the student needed to be reminded about the roles the developer, the computer, and the user play when constructing and executing a program.

The second week was dedicated to teaching the students about the IDE that was chosen for implementing the programming solutions. In our case, it was Visual Studio, and the applications that we built were business applications using Windows Forms projects. Here we explained the main features available to the students for designing, coding, and maintaining the applications they were building. In subsequent weeks, the focus was primarily on introducing the language syntax and building progressively complex applications that take advantage of the newly introduced language features. The classes are hands-on in nature, and the features introduced in the initial weeks continued to be used in subsequent weeks. In this course, we introduced students to business problems that require specific programming solutions and taught them to recognize which solutions corresponded to which problems. For example, students learned to distinguish between UI elements used for specific input and output situations, as well as to distinguish between the need to use in-memory collections versus visual lists. Students also learned to recognize user actions and learned how computers should respond to

actions such as button clicks or item selections from a list. The instructor always began by introducing concepts and providing supporting notes in order to make the requisite declarative knowledge accessible to students. Then students were asked on multiple occasions to apply what they had learned, in order to foster the development of procedural knowledge.

In this course, we started with simple calculator-based applications and built up to full CRUD applications using flat file systems with rudimentary data analysis capabilities. Along the way, students learned to use conditionals, loops, and collections, as well as operators and data types. We dedicated a single week to each new topic and added one more week for additional practice if the topic was complex. Our goal was to ensure that the newly introduced knowledge was converted into procedural knowledge and that the students could perform these newly introduced skills without the assistance of an instructor or a reference.

In addition to the classroom activities, we also gave students three other opportunities to repeat what they learned during each week. First, the students attended a separate hands-on lab session led by a teaching assistant. Second, the students were given optional assignments to solve over the weekend. Third, the students were given weekly quizzes that tested what they had learned the previous week. We continued with this approach of introducing new concepts and practicing and repeating solutions until the end of the semester.

Sample and Analysis

We take a quasi-experimental approach to answering our research questions (Cook & Campbell, 1979; Cook, Campbell, & Shadish, 2002), using the repetition strategy as it played out for a full year on the basis of five different sections of an undergraduate introductory programming course offered at a business school. We observed and assessed the progress of 72 undergraduate business students from the College of Business Administration (CBA) at Kuwait University (the sample characteristics in Table 1 below). The majority of students were females (72.2%), which is typical of universities in the GCC region (e.g., Malik & Coldwell-Neilson, 2018). While the introductory course is designed to be taken during a student's sophomore year, most students choose to postpone the course to their junior (40.28%) or senior years (38.89%) as expected (Woszczynski et al., 2005). Most of the students enrolled are Management Information Systems (MIS) majors (69.44%) followed by Operations Management (OM) majors (27.78%). Both majors can be considered non-computer science majors, given that CBA is classified

as a fine arts college and most admitted students do not receive training in math and programming in high school as would be expected from computer science majors.

Table 1
Sample Characteristics

	Count	%
Gender		
Male	20	27.78%
Female	52	72.22%
Academic Year		
Freshman	2	2.78%
Sophomore	13	18.06%
Junior	29	40.28%
Senior	28	38.89%
Major		
MIS	50	69.44%
OM	20	27.78%
Other	2	2.78%
Total	72	

The students were given a total of seven optional programming assignments over a period of six weeks, and were then assessed using a practical programming midterm exam. While we observed students throughout the semester, we noticed that other factors started to affect student performance during the second half of the semester-e.g., fatigue and workload pressures from other courses. We felt that the six-week period early in the semester offered enough time for students to practice before they became overwhelmed by the pressures of the semester, which introduced extraneous variability to our quasi-experiment.

We gave the students problems that required building practical business applications using VB.net. The problems required students to build event-driven Windows applications and to use language constructs such as variables, functions, and "if" statements. The students were offered optional assignments and asked to attend weekly lab sessions, as well as take weekly practical quizzes as further means of repetition. Since almost all students took a total of four quizzes, we excluded the number of quizzes taken as a study variable, given the lack of variability in the measure. The problems students worked on continued to progress in complexity every week and led up to a practical midterm exam that offered a larger problem that

was similar to the prior exercises that students had to solve within 50 minutes, without using any references other than assistance provided by the IDE. The student's grade in the midterm exam was used to assess the student's competency in programming-the logic being that the more proficient (i.e., automatic) students' problem-solving skills became, the more likely they would be to complete the requirements on time and earn a better grade. After completing the midterm, the students were asked to respond to a survey in which they self-reported some of the variables of the study. We obtained the remaining variables from standard class data collection procedures, such as lab attendance sheets and quiz grades. The variables used in the study, as well as some descriptive statistics, are summarized in Table 2:

Table 2
Study Variables and Descriptive Statistics

Variable	Description	Mean (STD)	Max	Min
Programming Competency (DV)	Practical midterm grade after 6 weeks of instructions	100.83 (32.3)	150	0
Assignment Repetitions	Number of optional assignments completed by the student	2.94 (2.11)	7	0
Lab attendance	Reported by TA	3.19 (2.49)	6	0
GPA	Self-reported	2.62 (0.55)	4	1
Gender	Self-reported categorical variable	M = 27.8% F = 72.2%		
Controls				
Age	Self-reported	22.35 (3.56)	37	19
Number of missed classes	Self-reported	1.39 (1.4)	6	0
Number of tardy classes	Self-reported	2.94 (2.11)	6	0
Times course was retaken	Self-reported	0.58 (0.95)	5	0

Once the data were collected, they were screened for errors and missing observations. We performed OLS regression using the statsmodels package (Seabold & Perktold, 2010), as the DV observed a relatively normal distribution (Figure 2) with no indication of violated assumptions or high influence from any single observation (Cohen, Cohen, West, & Aiken, 2003). We added interaction terms for both GPA and gender to assess whether the effect of repetition was affected by them (Aiken & West, 1991). We controlled for the effect of age, number of missed classes, number of tardy instances, and times a student retook the same course.

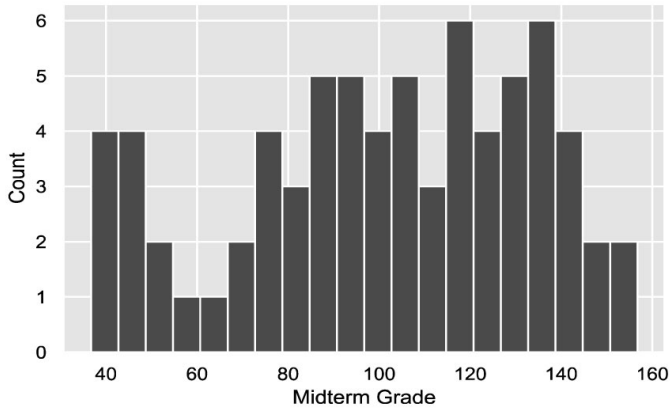

Figure 2: Distribution of Student Midterm Grades (DV)

Results

Table 3 shows the results of the OLS regression analysis. Our analysis was based on two models-the first is the main effects model that we use to test H1; the second is the interaction model used to test H2 and H3 to evaluate whether the effect of

Table 3
OLS Regression Analysis Results

Terms	Main Effect Model	Interaction Model
Repetition * Gender		2.761 (3.311)
Repetition * GPA		-3.914 (2.837)
Repetition	6.034 (1.759) **	15.149 (7.498) *
Lab Attendance	1.463 (1.295)	1.646 (1.31)
GPA	14.578 (6.161) *	28.813 (11.674) *
Gender (Female)	0.613 (7.376)	-9.09 (12.999)
Age	-1.707 (0.968)	-1.758 (0.978).
Missed Classes	-5.641 (2.584) *	-6.266 (2.612) *
Tardy Classes	0.191 (2.342)	-0.146 (2.357)
Retook Course	5.454 (3.648)	5.54 (3.698)
Intercept	44.487 (17.982) *	12.56 (29.749)
Number of Observations	72	72
R^2	42.9%	45.2%
Adjusted R^2	35.7%	36.2%

p-values: 0.05*, 0.01**, < 0.001***

repetition is affected by GPA or gender. The main effects model shows that the coefficient of assignment repetition was 6.034 (1.759), with significance at $p < 0.001$. This model suggests that for every bonus assignment completed, students' midterm grade increased by 6.034 points, even after accounting for other important factors such as student age, attendance, and GPA. The results suggest that using practical assignments based on the repetition strategy is an effective teaching tool that could improve all students' computer programming competency.

The other important variable related to repetition is lab attendance. While the effect of lab attendance is not significant, its model coefficient of 1.463 (1.295) suggests that midterm grades increased by 1.463 points for every lab a student attended. Given the small sample size, we cannot determine whether the effect is nonexistent, but the evidence suggests that further investigation is needed to determine the actual effect of repetition in terms of lab attendance. The small sample effect is compounded by the inconsistent lab attendance of 19 students, which represents a significant 23.4% of our sample (Figure 3). Therefore, we did not perform additional analysis on this variable or include interaction terms, as further analysis was not likely to yield any useable results. Therefore, we focused on assignment repetition as the main repetition variable.

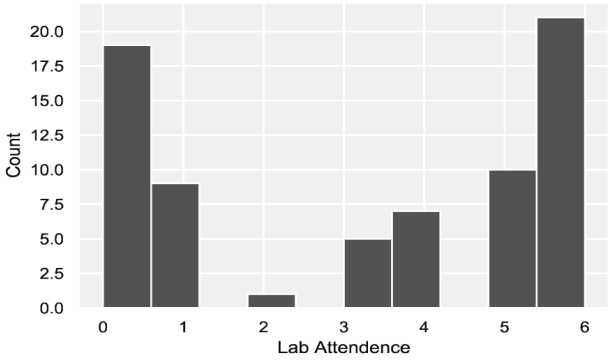

Figure 3: Histogram of Lab Attendance

One could argue that the significant relationship between assignment repetition and midterm grade may be skewed by students with higher GPAs, since those students would presumably be more likely to complete optional assignments. This possibility is especially significant given that GPA has a positive coefficient of 14.578 (6.161) in the model that is also significant. Therefore, to demonstrate that higher midterm grades are actually associated with repetition, rather than general

academic performance, it is important to test for H2 to see whether the positive effect of repetition also applies to students with lower GPAs.

For H2, we focus on the interaction model. Notice that the coefficient of interaction between the number of assignment repetitions and GPA is -3.914 (2.837)-which actually suggests the opposite of our expectation of the positive correlation between GPA and midterm grade. As the student's GPA increases, the effect of the optional bonus assignments on midterm performance diminishes by 3.914 for every point increase in GPA. While the effect is nonsignificant, we expect that this is due to the difficulties in detecting interaction effects from the residual variance after all the main effects have been accounted for (McClelland & Judd, 1993). While we cannot be certain that the effect exists based only on the p-value, other evidence suggests that the effect is likely there, but undetectable due to the small sample size.

To further illustrate what is happening, we breakdown the students into three groups based on their Grade Point Average (GPA) from the 4-point system to simplify the simple slope analysis. The first group (Group A) consists of above average students with GPA greater than 3.5. The second group (Group B) consists of average students with GPA between 3.5 and 2.5. The final group (Group C) consists of below average students with GPA less than 2.5. The swarm plot (Figure 4) shows how the values of assignment repetition for each student are distributed across different GPA and gender groups, which effectively rules out the claim that the benefits associated with repeating exercises is limited to students with high GPAs, or a specific gender, for that matter, as it shows no systematic distribution pattern.

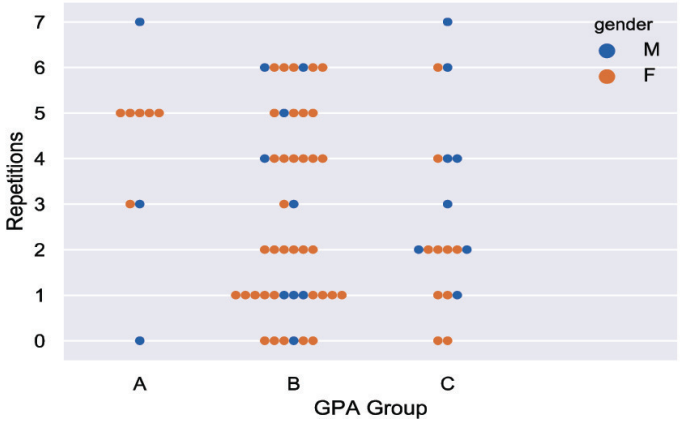

Figure 4: Swarm Plot Showing Distribution of Repetition Values Across GPA Categories and Gender Groups

To see if the effect of repetition changes by GPA group, we plot the simple slopes in Figure 5 for each of the GPA groups. Notice how the simple slope for group A students is almost flat, suggesting above average students will perform well on midterms irrespective of the times they repeat optional programming assignments. For group B students, the slope is steeper, suggesting that repetition might actually have a higher effect on midterm scores for average performing students. Similarly, the even steeper slope for group C students suggests that repetition might improve midterm scores even more for below average performing students.

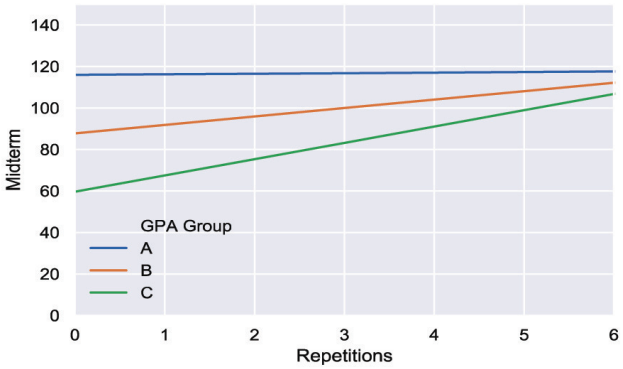

Figure 5: Simple Slope Plots for Different GPA Categories

These conclusions about the simple slopes cannot be confirmed without proper test statistics. We summarize the simple slope analysis and test statistics in Table 4 (Cohen *et al.*, 2003). The simple slope value for group A is 1.448 with a p-value of 0.673. Therefore, it cannot be concluded that it is different from zero and, as a result, it cannot be determined whether repetition has an effect on students in group A in our sample. The slope for group B is 5.362, which is significantly different from zero with a p-value of 0.012. Therefore, it can be concluded that assignment repetition has a significant and positive effect on students in group B. Yet, looking at the confidence intervals, it can be observed that the slope for group A and group B show some overlap in their confidence intervals; therefore, it cannot be concluded that the two slopes have a statistically significant difference between them. The observation that one of the slopes is statistically different from zero and the other one is not and that both slopes are not statistically different from each another is a strong indication that the interaction effect might indeed be there. However, it is difficult to detect given the small sample size and challenges of detecting interaction effect from residual variance (McClelland & Judd, 1993). It also explains why the interaction term in the OLS model is nearly significant. Similarly, with group C, the slope is 9.277, which is significant with a p-value of

0.012, as well. What the data suggests in regard to H2 is that for students from group A (i.e., GPA $\geqslant$ 3.5), the repetition strategy is not very important, as they will perform at high levels anyway. However, the results are important for B and C student groups (i.e., GPA $<$ 3.5). The results suggest that the repetition strategy is effective for these students and can improve their competency in computer programming. The results also strongly discount the earlier supposition that the effect of assignment repetition on midterm performance may be due to the efforts of high-performing students.

Table 4
Summary of Simple Slope Analysis and Test Statistics for GPA Categories

GPA Group	Simple Slope	SE	t-value	p-value	95% Confidence Interval	
					Lower	Upper
A	1.448	3.413	0.424	0.673	-5.377	8.273
B	5.363	2.0643	2.597	0.012	1.235	9.491
C	9.277	3.601	2.576	0.012	2.076	16.478

As for H3, the interaction term coefficient between gender and repetition is 2.76. While that coefficient would suggest that the effect of repetition is higher for the group of male students, relative to the group of female students, the effect is nonsignificant with a p-value of 0.487. This could also be attributed to the small sample size and the difficulty in detecting interaction effects in the residual variances (McClelland & Judd, 1993). The simple slopes are plotted in Figure 6 to illustrate the difference between the groups, which shows that the slope of the male group is slightly higher than the female group, but the difference is not significant, as we shall see in the details of the simple slope analysis.

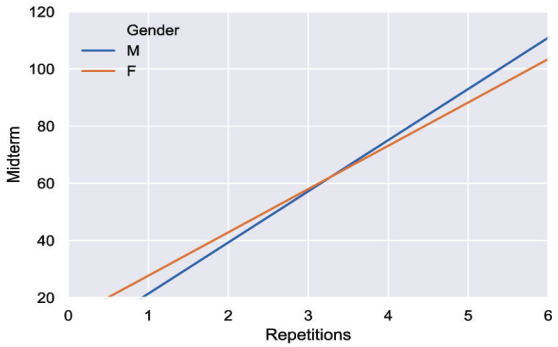

Figure 6: Simple Slope Plot for Different Gender Groups

The simple slope analysis is summarized in Table 5, which shows that repetition for both the male and female groups have simple slopes that are significantly different from zero. However, the difference between them is very small and nonsignificant, given the overlap between the confidence intervals of the two groups. The interaction term from OLS can also be viewed as a test of difference between simple slopes, because the interaction is between a continuous variable (i.e., Repetition) and a binary variable (i.e., Gender). Therefore, it can be concluded with respect to H3 that we were unable to detect significant differences in the effect of repetition between males and females. However, it cannot be concluded with certainty that there are no differences between how the two groups learn, due to the small sample of our study; therefore, further investigation using a larger sample size is warranted.

Table 5
Summary of Simple Slope Analysis and Test Statistics for Gender Groups

Gender	Simple Slope	SE	t-value	p-value	95% Confidence Interval	
					Lower	Upper
Males	7.909	7.896	2.268	0.027	2.12	33.698
Females	15.149	7.498	2.02	0.048	0.156	30.141

Discussion

Contributions

The results from our OLS analysis suggest that repetition, in the form of optional assignments that are similar to midterm challenges, has a significant impact on developing programming competencies for novices to be effective users of programming languages. The analysis also indicates that the repetition strategy is more effective for students with lower GPAs. A number of important theoretical contributions based on these findings can be highlighted.

First, computer programming is shown to be a deterministic cognitive task. As a result, in fields other than computer science, computer programming can be taught as an applied skill that can be used to solve specific problems. Through the repetition strategy, students are taught to recognize such problems and learn which programming constructs are most appropriate for building solutions. The findings also highlight the importance of deliberate practice in building competencies for the short term, and not just achieving expertise. They also present

a clear example of how different declarative programming knowledge (i.e., knowing about programming) is different from procedural programming knowledge (i.e., applying programming knowledge), and how the challenges faced by both students and instructors in acquiring or teaching each type of knowledge can be significantly different.

Second, we forward an empirical test that the power law of practice in the context of teaching computer novices computer programming is necessary. We are able to improve the skills of undergraduate business students by following a repetition strategy and have shown how the effectiveness of the strategy is related to a student's academic standing. The study also highlights the important role of instructors as a necessary condition for the success of this strategy, because they play an important role in resolving any misconceptions held by students about computer programming. This emphasizes a very important challenge that strategies not involving instructors will face and bring up a new line of inquiry about how detrimental student misconceptions about programming will be to such strategies.

Third, this work has some important practical contributions. Because repetition is key to improving competence in computer programming, it is recommended that programming classes be introduced early in curricula that use programming as a tool, not only for application development, but also for data analysis and visualization. The findings support the decision of a large proportion of IS undergraduate programs to keep introductory computer programming courses at the core of their programs (Bell, Mills, & Fadel, 2013) and not follow the suggestion from the latest AIS/ACM undergraduate IS curriculum guidelines to teach programming courses as electives (Topi *et al.*, 2010).

Fourth, the way the repetition strategy is implemented in this study shows how computer programming can be taught as an applied skill to students, provided the instructors be aware of student misconceptions of the programming task, as well as avoid cognitive overload when introducing new topics. Therefore, it can be argued that an applied approach that leverages simple languages and tools is likely to be fruitful, given that it is likely to reduce some of the cognitive burden on students when learning to program. In our case, the visual studio IDE seemed overwhelming for students at first, but they eventually learned to use only the features they needed to accomplish their tasks. Simpler development environments, such as the Jupyter Notebook, might reduce the time and effort needed by students to learn how to use the development tools and focus on problem solving. Once students learn to differentiate between programming and problem solving on the basis of

a single language or tool, they can more easily learn other tools and, perhaps, even other programming languages.

One final implication of this study is advice directed to fellow instructors, trainers, and students in fields familiar with the challenges and frustrations associated with teaching novices computer programming. There are no shortcuts-the key is patience and practice. A common occurrence when teaching computer programming is that some students will fail to recall the meaning of certain complex programming constructs, such as loops, conditionals, and functions. In fact, this is the rule rather than the exception, and it serves as a clear indication that the students need more practice through repetition before the knowledge related to these concepts is committed to long-term memory and easily recalled when needed. Instructors also need to be patient and more accepting of such expected shortcomings from students on their journey to becoming more competent programmers.

Limitations and future research

There are a number of limitations of this work that we hope to have been adequately addressed and could present opportunities for future research. First, this study involved a limited number of subjects-ideally, we would like to expand the scope of this study in the future. This limitation was not a choice made by the author of this manuscript but was a result of practical limitations as well as access to subjects. While this prevented us from detecting some effects, we believe that the effects we were able to detect in this small sample might be revealed to be greater in the context of a larger sample. Furthermore, our sample was limited to two business majors-namely, MIS and OM majors. While these students are technically inclined and, one could argue, may be similar in important ways to computer science majors, the business school in which we conducted this study is classified as a fine arts college. As such, most of these students have had limited prior exposure to math or programming. We confirmed this with a simple survey at the beginning of each semester, which indicated that most students had not been exposed to programming before the introductory course.

Another limitation is the confounding of some of the variables that could be interesting complements to this work. We therefore refrained from making concrete conclusions about these variables and pointed them out for purposes of potential future research. First, there is the effect of the programming language used. We were limited to VB.net and Visual Studio as a tool. While we noticed

students struggling with them at first, with enough practice they were able to master their use. It is suspected that using languages with less overhead, less boilerplate code, that are better designed for teaching (e.g., Python and Jupyter Notebook), might yield more impressive results. Furthermore, it would be interesting to explore how students handle learning multiple languages in a single course. Finally, there is the issue of generalizability of acquired student skills. Given that the intention of introductory courses is to endow students with highly specific skills, then we would argue that introductory courses have achieved their goals. However, it would be interesting to follow students that complete introductory programming courses to understand the factors that would make their newly acquired programming skill more generalizable.

Lab attendance was another important variable for which the data did not yield a useable result. We performed some comparisons between students that attended the lab sessions and students that did not. We did not find any systematic patterns that might have affected our findings. Both groups included students with varying midterm performance levels, and both groups also displayed varying interest in repetition assignments. It should be noted that the teaching styles of our various TAs may have affected variations in lab attendance. While there were no significant results for lab attendance, it should not be discounted as an important repetition variable, given our small sample size. In order to ensure proper examination of this variable, future studies could focus on lab attendance and possibly take into account potential reasons why students may not have attended the lab sessions.

Not exploring individual differences is one more limitation of this study. Such differences might explain the variability in student repetitions in terms of GPA and, perhaps, in gender. We also cannot make conclusions regarding the complexity of the programming tasks. We followed a preset protocol where one concept was introduced every week. Future research could examine how students would deal with unexpected problem complexity or how much of an increase in complexity students could handle in one week of instruction.

A recent discourse in computer programming education is related to whether programming should be taught as a standalone course focused on programming skills, or an integrated course where programming is used as a tool to solve domain specific problems (Schanzer, Krishnamurthi, & Fisler, 2019). The applied approach used in our study favors the integrated approach, yet the introductory

course is a dedicated programming course. In our discussions with students, we have noticed that students can be overwhelmed by the tools used to build the programs and be detracted from the main problems. An applied approach might give them more focus, as some of the students expressed that they had a better appreciation of computer programming after they had understood how it could be relevant to solving problems in their majors. Future research can explicitly study the effect of taking a more applied approach to programming and whether student learning ability is hindered if no dedicated programming course is offered.

Finally, based on our observing the aggregate effect of repetition by testing students performing a task with a specific time limit, future studies could take a more granular approach to assessing the time it takes to complete certain tasks and find whether repetition improves the speed with which the tasks are completed. Designing such an experiment for an actual class could be challenging. However, the results from this work are encouraging and may perhaps inspire funding for such a study in the future.

Conclusion

Can we train students or employees with no background in computer science to benefit from computer programming? We have shown, with empirical support, that repetition can be an effective strategy in training novices without computer science backgrounds to become competent users of programming languages. This finding was in fact amplified for students with lower GPAs. While answering this question, we have explained how computer programming is a cognitive skill that differs significantly from pure problem-solving skills. Our study indicates that students can improve computer programming skills through practice and perseverance while guided by an instructor. It is our hope that both instructors and students recognize the value of having such a skill in their toolbox even for business school majors. To meet the demands of this digital age in which information processing competency is increasingly important and also facilitates innovation and new venture creation that is crucial to economic development (Tomizawa, Zhao, Bassellier, & Ahlstrom, 2019), we hope that colleges and universities will introduce programming courses for all business majors. This should be at the beginning of their studies to give students ample opportunity to hone and apply these skills throughout their educational journeys and beyond.

References

Aiken, L. S., & West, S. G. 1991. *Multiple regression: Testing and interpreting interactions*. Thousand Oaks, CA, US: Sage Publications, Inc.

Anderson, J. R. 1982. Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369.

Anderson, J. R. 1992. Automaticity and the ACT theory. *The American Journal of Psychology*, *105*(2): 165-180.

Anderson, J. R. 2007. *How can the human mind occur in the physical universe?* New York, NY, US: Oxford University Press.

Anderson, J. R. 2015. *Cognitive psychology and its implications* (8th ed.). New York, NY, US: Worth Publishers.

Bauer, R., Mehrens, W. A., & Vinsonhaler, J. F. 1968. Predicting performance in a computer programming course. *Educational and Psychological Measurement*, *28*(4): 1159-1164.

Bell, C., Mills, R., & Fadel, K. 2013. An Analysis of Undergraduate Information Systems Curricula: Adoption of the IS 2010 Curriculum Guidelines. *Communications of the Association for Information Systems*, *32*(1). https://doi.org/10.17705/1CAIS.03202

Butcher, D. F., & Muth, W. A. 1985. Predicting performance in an introductory computer science course. *Communications of the ACM*, *28*(3): 263-268. https://doi.org/10.1145/3166.3167

Cohen, J., Cohen, P., West, S., & Aiken, L. 2003. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences* (Third Edition). New York, NY, US: Routledge.

Colley, A., & Comber, C. 2003. Age and gender differences in computer use and attitudes among secondary school students: What has changed? *Educational Research*, *45*(2): 155-165. https://doi.org/10.1080/0013188032000103235

Cook, T. D., & Campbell, D. T. 1979. The design and conduct of true experiments and quasi-experiments in field settings. In R. Mowday & R. Steers (Eds.), *Research in Organizations: Issues and Controversies*. Santa Monica, CA, US: Goodyear Publishing Company.

Cook, T. D., Campbell, D. T., & Shadish, W. 2002. *Experimental and quasi-experimental designs for generalized causal inference*. Boston, MA, US: Houghton Mifflin and Company.

DeKeyser, R. 2014. Skill Acquisition Theory. In B. VanPatten & J. Williams (Eds.), *Theories in Second Language Acquisition: An Introduction* (2nd ed.). New York, NY, US: Routledge.

Du Boulay, B. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1): 57-73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Ericsson, K. A., Krampe, R. T., & Tesch-Romer, C. 1993. The Role of Deliberate Practice in the Acquisition of Expert Performance. *Psychological Review*, *100*(3): 363-406.

Ericsson, K. A., Prietula, M. J., & Cokely, E. T. 2007. The making of an expert. *Harvard Business Review*, *85*(7/8): 114.

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

Forte, A., & Guzdial, M. 2005. Motivation and nonmajors in computer science: Identifying discrete audiences for introductory courses. *IEEE Transactions on Education*, *48*(2): 248-253.

Fowler, G. C., & Glorfeld, L. W. 1981. Predicting aptitude in introductory computing: A classification model. *AEDS Journal*, *14*(2): 96-109.

Good, J., & Howland, K. 2017. Programming language, natural language? Supporting the diverse computational activities of novice programmers. *Journal of Visual Languages & Computing*, *39*, 78-92. https://doi.org/10.1016/j.jvlc.2016.10.008

Hayashi, Y., Fukamachi, K.-I., & Komatsugawa, H. 2015. Collaborative Learning in Computer Programming Courses That Adopted the Flipped Classroom. *Proceedings of the 2015 International Conference on Learning and Teaching in Computing and Engineering*, 209-212. https://doi.org/10.1109/LaTiCE.2015.43

Karim, S., Carroll, T. N., & Long, C. P. 2016. Delaying Change: Examining How Industry and Managerial Turbulence Impact Structural Realignment. *Academy of Management Journal*, *59*(3): 791-817. https://doi.org/10.5465/amj.2012.0409

Kay, R. 2006. Addressing Gender Differences in Computer Ability, Attitudes and Use: The Laptop Effect. *Journal of Educational Computing Research*, *34*(2): 187-211. https://doi.org/10.2190/9BLQ-883Y-XQMA-FCAH

Konvalina, J., Wileman, S. A., & Stephens, L. J. 1983. Math Proficiency: A Key to Success for Computer Science Students. *Commun. ACM*, *26*(5): 377-382. https://doi.org/10.1145/69586.358140

Lahtinen, E., Ala-Mutka, K., & Jìrvinen, H.-M. 2005. A study of the difficulties of novice programmers. *Acm Sigcse Bulletin*, *37*(3): 14-18.

Lau, W. W. F., & Yuen, A. H. K. 2011. Modelling programming performance: Beyond the influence of learner characteristics. *Computers & Education*, *57*(1): 1202-1213. https://doi.org/10.1016/j.compedu.2011.01.002

Lowe, T. 2019. Explaining Novice Programmer's Struggles, in Two Parts: Revisiting the ITiCSE 2004 Working Group's Study Using Dual Process Theory. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*: 30-36. https://doi.org/10.1145/3304221.3319775

Malik, S. I., & Coldwell-Neilson, J. 2018. Gender differences in an introductory programming course: New teaching approach, students' learning outcomes, and perceptions. *Education and Information Technologies*, *23*(6): 2453-2475. https://doi.org/10.1007/s10639-018-9725-3

McClelland, G. H., & Judd, C. M. 1993. Statistical difficulties of detecting interactions and moderator effects. *Psychological Bulletin*, *114*(2): 376-390.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (Moti). 2013. Learning computer science concepts with Scratch. *Computer Science Education*, *23*(3): 239-264. https://doi.org/10.1080/08993408.2013.832022

Meyer, J., & Land, R. 2006. Threshold concepts and troublesome knowledge: An introduction. In J. Meyer & R. Land (Eds.), *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge*: 3-18. New York, NY, US: Routledge.

Moons, J., & De Backer, C. 2013. The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, *60*(1): 368-384. https://doi.org/10.1016/j.compedu.2012.08.009

Mow, I. T. C. 2008. Issues and Difficulties in Teaching Novice Computer Programming. In M. Iskander (Ed.), *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*: 199-204. Dordrecht, Netherlands: Springer.

Newell, A., & Rosenbloom, P. S. 1981. Mechanisms of skill acquisition and the law of practice. *Cognitive Skills and Their Acquisition*, 1: 1-55.

Newell, A., & Rosenbloom, P. S. 2013. Mechanisms of Skill Acquisition and the Law of Practice. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition*: 1-51. New York, NY: Routledge.

Newsted, P. R. 1975. Grade and Ability Predictions in an Introductory Programming Course. *SIGCSE Bulletin*, *7*(2): 87-91. https://doi.org/10.1145/382205.382897

Norman, D. A. 1987. Some observations on mental models. In R. M. Baecker & W. A. S. Buxton (Eds.), *Human-computer Interaction*: 241-244. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Nuutila, E., TÌ, S., & Malmi, L. 2005. PBL and Computer Programming-The Seven Steps Method with Adaptations. *Computer Science Education*, *15*(2): 123-142. https://doi.org/10.1080/08993400500150788

Pennington, N., & Grabowski, B. 1990. The tasks of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of programming*: 45-62. San Diego, CA, USA: Academic Press.

Perkins, D. N., & Martin, F. 1986. Fragile Knowledge and Neglected Strategies in Novice Programmers. *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*: 213-229. Retrieved from http://dl.acm.org/citation.cfm?id=21842.28896

Perkins, D. N., Schwartz, S., & Simmons, R. 1988. Instructional strategies for the problems of novice programmers. In *Teaching and learning computer programming: Multiple research perspectives*: 153-178. Hillsdale, NJ, US: Lawrence Erlbaum Associates, Inc.

Rist, R. S. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and intermediate Student Programmers. *Human-Computer Interaction*, *6*(1): 1-46. https://doi.org/10.1207/s15327051hci0601_1

Robins, A. 2019. Novice Programmers and Introductory Programming. In S. Fincher & A. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (1st ed.: 327-376). https://doi.org/10.1017/9781108654555.013

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

Rogalski, J., & Samurçay, R. 1990. Acquisition of programming knowledge and skills. In J.-M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of programming*: 157-174. San Diego, CA, USA: Academic Press.

Rubio, M. A., Romero-Zaliz, R., MaC., & de Madrid, A. P. 2015. Closing the gender gap in an introductory programming course. *Computers & Education*, 82: 409-420. https://doi.org/10.1016/j.compedu.2014.12.003

Schanzer, E., Krishnamurthi, S., & Fisler, K. 2019. What does it mean for a computing curriculum to succeed? *Communications of the ACM*, *62*(5): 30-32. https://doi.org/10.1145/3319081

Seabold, S., & Perktold, J. 2010. Statsmodels: Econometric and statistical modeling with python. *9th Python in Science Conference*.

Serrano-Cmara, L. M., Paredes-Velasco, M., Alcover, C.-M., & Velazquez-Iturbide, J. ê. 2014. An evaluation of students' motivation in computer-supported collaborative learning of programming concepts. *Computers in Human Behavior*, 31: 499-508. https://doi.org/10.1016/j.chb.2013.04.030

Singley, M. K., & Anderson, J. R. 1989. *The transfer of cognitive skill*. Cambridge, MA, USA: Harvard University Press.

Sorva, J. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education*, *13*(2): 8:1-8:31. https://doi.org/10.1145/2483710.2483713

Striegel, A., & Rover, D. T. 2002. Problem-based learning in an introductory computer engineering course. *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, *2*, F1G-F1G. IEEE.

Tomizawa, A., Zhao, L., Bassellier, G., & Ahlstrom, D. 2019. Economic growth, innovation, institutions, and the Great Enrichment. *Asia Pacific Journal of Management*. https://doi.org/10.1007/s10490-019-09648-2

Topi, H., Valacich, J. S., Wright, R. T., Kaiser, K., Nunamaker Jr, J. F., Sipior, J. C., & de Vreede, G.-J. 2010. IS 2010: Curriculum guidelines for undergraduate degree

programs in information systems. *Communications of the Association for Information Systems*, *26*(1): 18.

Wagner, I. 2016. Gender and Performance in Computer Science. *ACM Transactions on Computing Education*, 16(3): 11:1-11:16. https://doi.org/10.1145/2920173

Wilson, B. C., & Shrock, S. 2001. Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors. *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, 184-188. https://doi.org/10.1145/364447.364581

Winslow, L. E. 1996. Programming pedagogy-A psychological overview. *ACM SIGCSE Bulletin*, *28*(3): 17-22. https://doi.org/10.1145/234867.234872

Woszczynski, A., Haddad, H., & Zgambo, A. 2005. *An IS Student's Worst Nightmare: Programming Courses*. 5.

الملخص

التكرار كوسيلة لتعليم المبتدئين: برمجة الحاسب الآلي في كليات التجارة والعلوم الإدارية

محمد نجيب المرزوق
جامعة الكويت

هدف الدراسة: دراسة فاعلية أسلوب التكرار في تعليم المبتدئين من غير تخصصات علوم الحاسوب: برمجة الحاسب الآلي.
منهجية الدراسة: الدراسة مصممة كشبه تجربة وتستخدم الانحدار الخطي البسيط مع الرسم البياني للميل في تحليل النتائج.
البيانات وعينة الدراسة: العينة تحتوي على 72 طالب بكالوريوس علوم إدارية من مقرر أولي في تعليم برمجة الحاسوب.
نتائج الدراسة: وجود علاقة طردية معتبرة بين تكرار حل التكاليف البرمجية الاختيارية مع تطور مهارة برمجة الحاسوب لدى الطالب بشرط وجود موجه للعملية التعليمية، كما لوحظ زيادة قوة هذه العلاقة حال تدني أداء الطالب أكاديمياً.
الخاتمة: إن أسلوب التكرار في تعليم برمجة الحاسوب مع وجود الموجه أسلوب فعال خصوصاً لمن كان أداؤهم الأكاديمي متدنيًا، كما تبرز الدراسة أهمية الوقت في تعلم برمجة الحاسوب، وأهمية إدراج مقررات تعلم برمجة الحاسب الآلي مبكراً ضمن مناهج تخصصات كليات التجارة والعلوم الإدارية.

Mohammad N. AlMarzouq is an assistant professor of information systems at Kuwait University. He received his PhD in management information systems from Clemson University. His research interests include Free/Libre and Open Source software, project management, technology acceptance, diffusion of innovation, as well as computer programming education and training. His work has been published in Decision Support Systems, Information and Management, Computers in Human Behavior, and Communications of the AIS. (mohammad.almarzouq@ku.edu.kw)

مجلة دراسات الخليج والجزيرة العربية

مجلة علمية فصلية محكمة تصدر عن مجلس النشر العلمي - جامعة الكويت
صدر العدد الأول منها في يناير عام 1975م

رئيس التحرير

أ. د. عبدالعزيز غانم الغانم

ترحب المجلة بنشر البحوث والدراسات العلمية المتعلقة بشؤون منطقة الخليج والجزيرة العربية في مختلف علوم البحث والدراسة .

ومن أبوابها

- البحوث العربية.
- البحوث الإنجليزية.
- عرض الكتب ومراجعتها .
- البيبلوجرافيا العربية.
- البيبلوجرافيا الإنجليزية.
- ملخصات الرسائل الجامعية: (ماجستير - دكتوراه).
- التقارير: مؤتمرات - ندوات

الاشتراكات

ترسل قيمة الاشتراك مقدماً بشيك لأمر - جامعة الكويت
مسحوب على أحد المصارف الكويتية

داخل دولة الكويت : للأفراد : 3 دنانير - للمؤسسات : 15 ديناراً
الدول العربية : للأفراد : 4 دنانير - للمؤسسات : 15 ديناراً
الدول الغير عربية : للأفراد : 4 دنانير - للمؤسسات : 15 ديناراً

توجه جميع المراسلات باسم رئيس تحرير مجلة دراسات الخليج والجزيرة العربية

ص.ب: 17073 الخالدية - الرمز البريدي: 72451 الكويت
تلفون: 24984067 - 24984066 - 24833215 965+ - فاكس: 24833705 965+
E-mail: jgaps@ku.edu.kw - http://www.pubcouncil.kuniv.edu.kw/jgaps

تفهرس المجلة وملخصاتها ونصوصها في قواعد البيانات والمعلومات التالية:
EBSCO Publishing Products
دار المنظومة: www.mandumah.com
الرقم الدولي للمجلة: ISSN : 0254 - 4288