Worked-out Examples in a Computer Science Introductory Module

Isabel C. Moura

Abstract—**Active learning engages students in Computer Science (CS) classroom activities. However, instructing novice undergraduates to solve programming problems primarily by making them generate the solutions puts a heavy load on these students' working memory, preventing them from learning CS fundamentals. This appears to have happened in the 2009 edition of a CS introductory module at the University of Minho, given its high failure rates. Studying CS worked-out examples directs novice undergraduates' attention to learning the essential of relations between problem-solving moves, diminishing their working memory load. This pilot study describes how worked-out examples have been successfully applied to the 2010 edition of that same CS introductory module, reducing its failure rates. The study portrays the changes in this module from a program-generation to a program-completion approach. The results in terms of success, failure, and dropout of a more program-completion approach are given and analyzed.**

Index Terms—**computer science education, novice programmers, programming, worked-out examples**

I. Introduction

At the University of Minho (UM), School of Engineering, students who choose to graduate in Polymers Engineering Integrated Master (PEIM), which is a five-year degree program, have to pass the two-module Programming and Numerical Methods (PNM9703) course. Programming is a Computer Science (CS) introductory module of this second year course of PEIM studies. For these novice undergraduates learning CS fundamentals entails acquiring and developing complex knowledge and practical skills, such as, reading and writing programs [1], [3], [6], [10], [15]. But, expecting these students to solve programming problems mainly by generating programs may have led to the high failure rate associated to the programming module of the PNM9703 course in the year of 2009.

The referred module, offered by the Department of Information Systems, covers two thirds of the semester. So, the material taught consisted of programming basic constructs (e.g., variables, assignment statements, selections, loops, and arrays). In-class active instructional activities were used to introduce these CS fundamental skills and concepts. That is, during each lab session the instructor presented a basic programming problem and led the students to build the corresponding algorithmic solution for them to code, test, and debug [2], [7], [11].

In the 2009 edition of the programming module students were further assigned a set of basic programming problems (to solve both in class and at home on a weekly basis) so they could apply those skills and concepts by building, coding, testing, and debugging their own solutions. Evaluation, which consisted of two individual tests, also required the students to write down the algorithmic solution and the corresponding code for a basic programming problem. But, the reported instructional and learning activities failed to positively influence students' programming achievements in this semester.

Research in education and cognitive load theory suggests that, before being able to write a piece of code, novice undergraduates should be able to read it. Furthermore, both the abilities of reading and writing programs should be taught. However, the complexity of these skills may impede the learning of novice undergraduates due to the limited capacity of the human's working memory when dealing with new information. So, teachers should organize the programming knowledge in order to decrease these students' working memory load. For instance, students may start being given the complete solutions to programming problems (or worked-out examples) and then complete increasingly larger parts of incomplete, given solutions. Studying worked-out examples further directs students' attention to learning the essential of relations between problem-solving moves. The more students learn to recognize which moves are required for particular programming problems, the more likely they are to succeed in learning CS fundamentals (e.g., [4]–[6], [8]–[10], [13], [15]).

In the 2010 edition of the programming module, students were handed over a set of algorithmic solutions that solved basic programming problems. These solutions were short, textbook-type algorithmic segments of 1 to 30 lines long (tops) which started by being complete and flawless. As the weeks progressed, flaws and missing lines were increasingly added to those solutions for students to complete and/or correct. Throughout the programming module of the PNM9703 course students were supposed to read, complete, and/or correct each algorithmic solution and to code, test, and debug it. They were further requested to find out the purpose of the given solution (i.e., the basic programming problem). The two individual tests consisted mainly of multiple-choice questions. These tests aimed at evaluating students' recognition of syntactic errors and understanding of the structure and function of simple algorithmic and code

Manuscript received October 03, 2011; revised November 09, 2011.
I. C. Moura is with the Centro Algoritmi, Universidade do Minho, 4800-058 Guimarães, Portugal (phone: 351-253-510266; fax: 351-253-510300; e-mail: icm@dsi.uminho.pt).

sequences [15].

This pilot study reports on the impact that the worked-out examples implementation of the PNM9703 programming module (at the UM in the fall semester of 2010) had on its failure rate and students' academic achievements. Method and results are then presented. A discussion follows on this pilot study's results and potential ways of improving such an implementation.

II. The Learning Environment

It is generally accepted, at the UM, that programming is a useful skill for students not majoring in CS. This is the main reason why the latter topic is part of the PEIM studies. For the second year of these studies, the PNM9703 fall semester course comprises two modules–the programming module and the numerical methods module. The course is mandatory and has no prerequisites. The programming module lasts for two thirds of the fall semester and focuses in the initial development of programming skills. However, these skills are hard to learn in such a short period of time. Thus, the CS material taught was reduced to its basic constructs (e.g., variables, assignment statements, selections, loops, and arrays) to avoid over expose students to content through the instructor. This would help them have sufficient opportunity to interact with the content in a meaningful way and learning would not be blocked [14].

Both the 2009 and the 2010 editions of the programming module consisted of a weekly 130-minute lab session. In each session students were introduced to a basic programming problem (refer to Appendix A). Then, first, they were lectured, for approximately 5-15 minutes, on the algorithmic constructs intended for the solution of that problem. Second, students were asked to put together (individually or in groups of two) an algorithmic solution for the given problem in a couple of minutes (i.e., students tried to apply the knowledge lectured). Third, one of the students' solutions was written, discussed, and improved on the board by having the instructor asking 'what-if' questions and students being given time to work on their answers and presenting them before class. Finally, using their personal computers or the lab ones, students were guided through the programming language text book so they could code, test, and debug (individually or in groups of two) the algorithmic solution discussed in class [1], [2], [5], [7], [11]. Visual Basic was the adopted programming language. The MS Excel 2007 VBA programming environment was chosen to make it easy for students to automate the handling of datasheets they work with throughout the PEIM studies. The assessment of students' achievements consisted of two individual tests.

In the fall semester of 2009, the learning of CS fundamentals in the programming module of the PNM9703 course was expected to occur mainly through the practice of generating programs to solve basic problems. But, this approach seems to have had a negative influence on many of the students. Although in-class active instructional and learning techniques were used, the bulk of students' activities involved solving a set of basic programming problems both in class and at home on a weekly basis, with the instructor providing hints and directions as needed. That is, students were supposed to come up with an algorithmic solution and the corresponding code for each given basic programming problem. Both hand-written tests evaluated students individually on these same sorts of abilities. Overall grades of the 2009 programming module of the PNM9703 course were derived 50% from each test.

Expecting novice undergraduates to learn CS fundamentals by making them solve problems can be ineffective for some of them [4], [6], [9], [10], [13]. Thus, in the fall semester of 2010, the instructor started each lab session putting into practice the above referred in-class active instructional and learning activities (and revisiting CS fundamentals previously taught as needed). The remainder of the class was used to make undergraduates study a couple of algorithmic solutions (or worked-out examples) that solved basic programming problems. These solutions were textbook-type algorithmic segments (of 1 to 30 lines long, tops) applying the basic construct(s) intended for a particular lab session (see Appendix B). The solutions were made available to students on the course web site the week before the session took place. Throughout the module flaws and missing lines were increasingly added to the given algorithmic solutions. Students were supposed to study, complete, and/or correct them, plus, code, test (by compiling and running the programs), and debug these solutions under MS Excel 2007 VBA programming environment. The instructor wandered around giving advice and directions as needed. Having both the error-free algorithmic solution and the corresponding coded one, students were asked to summarize the function of it (i.e., somehow, to find the basic programming problem being solved). At home, and on a weekly basis, students were supposed to finish the worked-out examples started in class. Many novice undergraduates are unable to write a piece of code by the end of a whole semester practicing programming [6]. So, the first test consisted entirely of multiple-choice questions (of four options each) to evaluate students' recognition of syntactic errors and understanding of the structure and function of simple algorithmic and code sequences [15]. The second test further required students to fill-in the gaps for a given simple algorithmic segment and to write a simple piece of code equivalent to a given one. Overall grades of the 2010 programming module of the PNM9703 course were derived 40% from the first test and 60% from the second test.

III. Research Questions and Method

Research in education and cognitive load theory suggests that instructors shall organize the programming knowledge so novice undergraduates start by reading (and further completing) worked-out examples of programs instead of generating them to solve problems. Studying worked-out examples directs students' attention to learning the essential of relations between problem-solving moves and reduces students' working memory load. The more students learn to recognize which moves are required for particular programming problems, the more likely they are to succeed in learning CS fundamentals (e.g., [4]–[6], [8]–[10], [13], [15]). This hypothesis raised the following research questions:

1) Are there differences in the programming module approval, failure, and dropout rates between the edition of the PNM9703 course taught in the fall of 2010 and the one taught in the fall of 2009?
2) Are there differences in approved students' final programming achievements (on average) for the programming module between the edition of the PNM9703 course taught in the fall of 2010 and the one taught in the fall of 2009?

In the current study quantitative methodologies were used in the analysis and interpretation of data.

IV. Results

Data spanning two semesters were collected. The students are from the PEIM degree program. A few of them were excluded from the sample of the fall semester of 2010 because they have already been taught a similar content in the fall of 2009, and thus, the improvement in their grades is expected. Overall, data from a total of 77 students were examined. In the fall semester of 2010, the 28 sample students registered in PNM9703 were enrolled in the second year of PEIM studies. But, this was not the case in the fall semester of 2009 (refer to Table I, where data from six students is missing regarding the '*Year*' variable): two students were enrolled in the fifth year of PEIM studies, three in the fourth year, seven in the third year, and the remaining 31 in the second year of these same studies. The undergraduates participating in this study had different backgrounds since statistically significant differences between semesters were found in the distribution of the students' academic index (Z = -3.03, *p-value < 0.01*). Because of their higher academic index, it is expected that the students who attended the 2009 edition of the PNM9703 course would be in a better position to learn CS fundamentals mainly by generating programs and to perform better on tests compared to the colleagues who attended the 2010 edition. In fact, all the fifth year undergraduates got approval in the 2009 programming module. But, none (out of three) of the fourth year students were approved and only two (out of seven) of the third year students passed this same module.

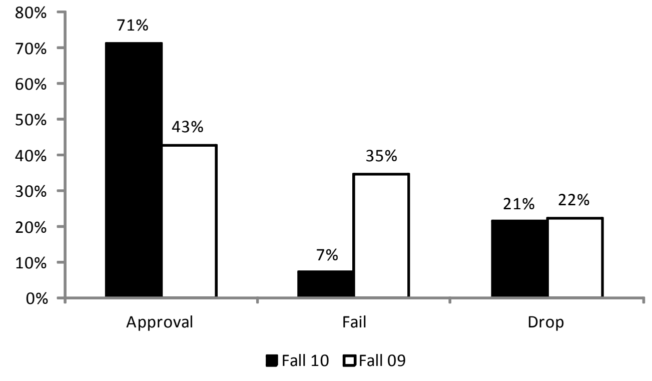

Fig. 1. Approval, failure, and withdrawal rates on the programming module of the PNM9703 course in the fall of 2010 and in the fall of 2009.

In the fall semester of 2010, out of 28, 20 students were approved (i.e., 71% of approvals) and two students were failed (i.e., 7% of failures). The remaining six students dropped the programming module of the PNM9703 course (i.e., 21% of withdrawals). According to Fig. 1, the programming module failure rate of this semester indicates that undergraduates might have responded favorably to the worked-out examples implementation of this particular PNM9703 course module. For students involved in this approach: the failure rate was five times lower (7%) than the one of the fall semester of 2009; and the approval rate almost doubled (71%) the one of the fall semester of 2009. The drop rates were about the same in both semesters.

TABLE I
The Students' Distribution (in Percentage) by Academic Year

Year	Fall 2010 (N = 28)	Fall 2009 (N = 43)
2nd	100	72
3rd	-	16
4th	-	7
5th	-	5

Regarding the first research question, the test result (z = -2.74, *p-value < 0.01*) for the proportion of failures indicates that the failure rate of the programming module in the 2010 edition of the PNM9703 course (N = 28) is numerically and statistically different from the failure rate of the fall semester of 2009 (N = 49). A similar test performed on the proportion of approvals of the 2010 programming module reached a similar result (z = 2.37, *p-value < 0.05*).

Concerning students' achievements, the final programming grade average for the approved ones was equal to 12 (SD = 2.06; Maximum = 16; Minimum = 10; N = 20), on a 0-20 scale, in the fall semester of 2010. In the fall of 2009, the final programming grade average for the approved students of the PNM9703 class equaled the same mark (SD = 2.19; Maximum = 19; Minimum = 10; N = 21). Examining the second research question, no numerically and statistically significant differences between semesters were found in the distribution of the approved students' final programming grades.

V. Discussion, Conclusions, and Recommendations

This pilot study reports on a worked-out examples implementation in an undergraduate CS introductory module of the PNM9703 course (i.e., the programming module offered by the Department of Information Systems) at the UM in the fall of 2010. This module implementation comprised:

1) In-class active instructional and learning activities.
2) Studying and, later on, completing and/or correcting short, textbook-type algorithmic solutions for basic programming problems (or worked-out examples that were handed over complete and flawless, in the beginning, and increasingly incomplete and/or flawed as the module progressed).
3) Coding, testing, and debugging the referred algorithmic solutions.
4) Two individual test assignments consisting of, mainly, multiple-choice questions.

The results of the 2010 programming module implementation indicate that students responded favorably to the worked-out examples approach. That is, given Fig. 1 results, the referred implementation of the programming module of the PNM9703 course (in the fall semester of 2010) provided a better learning environment for novice undergraduates to recognize which moves are required to solve basic programming problems; and it might have reduced students' working memory load compared to the implementation of the fall semester of 2009. Thus, the more likely students were to succeed in learning CS fundamentals [4]–[6], [8]–[10], [13], [15]. Still, with respect to those students who passed, the PNM9703 course programming module final grade average for the fall semester of 2010 equaled (reaching the 12 mark, on a 0-20 scale) the final grade average for the fall semester of 2009. The statistically significant differences concerning the students' backgrounds between semesters, with the supposedly most programming experienced ones taking the latter programming module, may explain this result [7], [16].

Active learning techniques may keep students highly engaged in generating programs. But, the heavy load on novice undergraduates' working memory that the latter implies prevents some of them from learning CS fundamentals [4], [6], [8]–[10]. The reported worked-out examples implementation may mitigate this problem as long as the instructor motivates the students to study the examples. This can be done by alternating an algorithmic solution (or worked-out example) with the coding, testing, and debugging of it. Plus, reminding students (throughout the semester as needed) that they will be tested on the understanding of the structure and function of algorithmic and code sequences structurally identical to the given examples [12]. Furthermore, the algorithmic solutions shall, somehow, resemble the code intended for that solution, since novice undergraduates have a hard time going between different representations of the same solution. That is, appropriately constructed worked-out examples shall avoid rising the cognitive load on these students' working memory [6].

Appendix A: an Example of a Basic Programming Problem

Build a program that computes the average of three given grades for a student.

Appendix B: an Example of an Algorithmic Solution

Version 1.1	

grade1, grade2, grade3: REAL;
sum, average: REAL;
	declaration of variables

Read(grade1); Read(grade2); Read(grade3);	data input

sum $\leftarrow$ grade1 + grade2 + grade3;
average $\leftarrow$ sum / 3;
	computation

Write(average).	data output

References

[1] M. Barak, J. Harward, G. Kocur, and S. Lerman, "Transforming an introductory programming course: from lectures to active learning via wireless laptops," *Journal of Science Education and Technology*, vol. 16, no. 4, pp. 325–336, 2007.

[2] R. Felder and R. Brent. (2009). Active learning: an introduction. *ASQ Higher Education Brief* [Online]. *2(4)*. Available: http://www.asq.org/edu/2009/08/best-practices/active-learning-an-introduction.%20felder.pdf

[3] A. Forte and M. Guzdial, "Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses," *IEEE Transactions on Education*, vol. 48, no. 2, pp. 248–253, 2005.

[4] P. Kirschner, J. Sweller, and R. Clark, "Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching," *Educational Psychologist*, vol. 41, no. 2, pp. 75–86, 2006.

[5] M. Linn and M. Clancy, "The case for case studies of programming problems," *Communication of the ACM*, vol. 35, no. 3, pp. 121–132, 1992.

[6] R. Lister, "After the gold rush: toward sustainable scholarship in computing," in *Proc. 10th Conference on Australasian Computing Education*, Wollongong, 2008, pp. 3–17.

[7] J. McConnell, "Active learning and its use in computer science," in *Proc. 1st Conference on Integrating Technology into Computer Science Education*, Barcelona, 1996, pp. 52–54.

[8] M. Prince and R. Felder, "The many faces of inductive teaching and learning," *Journal of College Science Teaching*, vol. 36, no. 5, pp. 14–20, 2007.

[9] M. Prince and R. Felder, "Inductive teaching and learning methods: definitions, comparisons, and research bases," *J. Engr. Education*, vol. 95, no. 2, pp. 123–138, 2006.

[10] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: a review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[11] K. Smith, S. Sheppard, D. Johnson, and R. Johnson, "Pedagogies of engagement: classroom-based practices," *J. Engr. Education*, vol. 94, no. 1, pp. 87–101, 2005.

[12] J. Sweller and G. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, 1985.

[13] J. van Merriënboer, P. Kirschner, and L. Kester, "Taking the load off a learner's mind: instructional design for complex learning," *Educational Psychologist*, vol. 38, no. 1, pp. 5–13, 2003.

[14] M. Weimer, *Learner-Centered Teaching. Five Key Changes to Practice.* San Francisco, CA: Jossey-Bass, 2002.

[15] S. Wiedenbeck, "Novice/expert differences in programming skills," *International. Journal of Man-Machine Studies*, vol. 23, no. 4, pp. 383–390, 1985.

[16] B. Wilson and S. Shrock, "Contributing to success in an introductory computer science course: a study of twelve factors," *ACM SIGCSE Bulletin*, vol. 33, no. 1, pp. 184–188, 2001.