New Orleans JAZZED about Engineering Education

Paper ID #16585

In-code Comments as a Self-explanation Strategy for Computational Science Education

Mr. Camilo Vieira, Purdue University

PhD Candidate at Purdue University Master of Engineering in Educational Technologies - Eafit University Systems Engineer - Eafit University

Dr. Anindya Roy, Johns Hopkins University
Dr. Alejandra J. Magana, Purdue University, West Lafayette

Alejandra Magana is an Associate Professor in the Department of Computer and Information Technology and an affiliated faculty at the School of Engineering Education at Purdue University. She holds a B.E. in Information Systems, a M.S. in Technology, both from Tec de Monterrey; and a M.S. in Educational Technology and a Ph.D. in Engineering Education from Purdue University. Her research is focused on identifying how model-based cognition in STEM can be better supported by means of expert technological and computing tools such as cyber-physical systems,visualizations and modeling and simulation tools.

Prof. Michael L. Falk, Johns Hopkins University

Michael Falk is a Professor in the Department of Materials Science and Engineering at Johns Hopkins University's Whiting School of Engineering where he has served on the faculty since 2008 with secondary appointments in Mechanical Engineering and in Physics and Astronomy. He holds a B.A. in Physics (1990) and a M.S.E. in Computer Science (1991) from Johns Hopkins University and a Ph.D. in Physics (1998) from the University of California, Santa Barbara. He has been twice selected as a visiting Chaire Joliot at the École Supérieure de Physique et de Chimie Industrielles at Paris Tech and has organized extended workshops on the physics of glasses and on friction, fracture and earthquakes at the Kavli Institute for Theoretical Physics. He has received several awards for his educational accomplishments, and in 2011 he received an award from the university's Diversity Leadership Council for his work on LGBT inclusion. His education research focuses on integrating computation into the undergraduate core curriculum. Falk also serves as the lead investigator for STEM Achievement in Baltimore Elementary Schools (SABES) an NSF funded Community Enterprise for STEM Learning partnership between JHU and Baltimore City Schools.

Dr. Michael J. Reese Jr., Johns Hopkins University

Michael Reese is the Associate Director at the Johns Hopkins Center for Educational Resources. Reese previously worked as an Educational Technologist at Caliber Learning and Booz-Allen and Hamilton. He also consulted with the University of Maryland School of Nursing on the launch of their distance education program. He earned a Ph.D. in sociology from the Johns Hopkins University and an M.Ed. in educational technology from the University of Virginia. He graduated with a B.S. in electrical engineering at Virginia Tech, where he was named the Paul E. Torgersen Leadership Scholar.

In-code Comments as a Self-explanation Strategy for Computational Science Education

Abstract

Computational science and engineering is an important field that integrates computational tools and methods along with and disciplinary sciences and engineering to solve complex problems. However, several research studies and national agencies report that engineering students are not well prepared to use or create these computational tools and methods in the context of their discipline. Furthermore, some of the skills within computational science and engineering (e.g., programming) can be difficult to learn. This study explores potential pedagogical strategies for the implementation of worked-examples in the context of computational science and engineering education. Students' self-explanations of a worked-example are collected as in-code comments, and analyzed to identify effective self-explanation strategies. The results from this study suggest that students' in-code comments: (1) can be used to elicit self-explanations and engage students in exploring the worked-example; and (2) show differences that can be used to identify the self-explanation effect.

Background and Motivation

Several reports have suggested that there are not enough professionals with the appropriate skills to use computation in order to solve complex problems in their fields [1-3]. Higher education should respond to this need to prepare future engineers and scientists with the skills to use and create computational tools and methods. A challenge is that computer science concepts have been traditionally taught separately from disciplinary concepts. Students often learn programming from a computer science perspective, but they may not know how to apply these methods and techniques to solve problems in their discipline[3].

Some universities and engineering departments have initiated curricular innovations to introduce computational science within the engineering curricula. For instance, the materials science and engineering department at a large Mid-Atlantic university created a specific programming course for materials science undergraduate students[4]. This department also created and integrated several computational modules into the six materials science core courses to support solving disciplinary problems. This presented an opportunity to conduct research to identify effective pedagogical strategies to scaffold student learning in this context [4,5]. Identifying appropriate scaffolding methods in this endeavor is important because learning computer programming is a complex task in itself. Now, pairing programming with disciplinary concepts may increase the complexity of this learning process[4].

To scaffold the integration of programming concepts with disciplinary concepts, this study explores student in-code comments as a self-explanation strategy for a given worked-example. The context of the study is a materials science and engineering programming course that involves in-class programming activities, as well as five computational projects on disciplinary problems. The guiding research questions are:

- *What are the characteristics of student self-explanations of programming worked examples involving a disciplinary context?*
- *How are student's self-explanation characteristics related to their performance in a computational project involving a disciplinary context?*

Background Literature

Computer programming is a complex skill to learn[6]. When novice programmers start learning to code, they need to consider many different elements at once. The programming logic, language syntax and semantics, structure of the program, and purpose of the program are just some of the many elements students need to consider simultaneously[6, 7]. Moreover, introducing disciplinary problems as part of a programming task may bring additional level of complexity.

Cognitive load theory (CLT)[8] can explain how different forms of complexity can affect learning. CLT uses a cognitive architecture and the characteristics of the material to describe how learning occurs. The cognitive architecture consists of a limited working memory and a vast long-term memory. When learning occurs, the working memory processes all the elements from the instructional material before creating a schema that is stored in the long-term memory. Learning all these interacting elements at once can produce a cognitive overload in students' processing. Thus, an appropriate scaffolding approach is important to avoid cognitive overload. For example, visual programming languages remove the syntax errors from the equation by providing drag and drop building blocks[9]. Also, modeling and simulation have been paired with problem solving pedagogies to support student learning, both at the undergraduate[10] and the graduate[11] levels. Organizing the tasks as a step-by-step problem-solving procedure while taking advantage of computational representations may benefit the student-learning process.

A common pedagogical approach for computational science education is known *as use-modify-create*[12]. This approach suggests that it is important to first allow learners to familiarize themselves with an existing solution. Once learners have acquired some understanding and skills, it is time for them to make some changes in the solution. The final step in the learning process would involve creating a complete solution themselves.

The completion effect[8] is a pedagogical approach similar to use-modify-create approach. The idea behind the completion effect is to increasingly remove scaffolding as students learn. Adaptive Control of Thought-Rational (ACT-R)[13] is a framework for skill acquisition that provides guidelines for the implementation of the completion effect in a four-step approach. The first step consists of having students solve problems by making analogies from worked examples. This process allows the learner to acquire basic declarative rules of the complex task. The second step consists of understanding the basic principles of the problems in that context. During the third stage students practice problem solving so that the procedural rules become evident. Finally, students create schemas in the long-term memory that will allow them to solve different problems without using the worked examples.

In order to be effective, the worked examples need to follow certain instructional principles[14]. A worked example should include: (1) a problem statement; (2) a procedure for solving the problem; and (3) auxiliary representations of a given problem. The recommended features of these components are:

- *Intra-example features*: the worked examples should be designed using multiple formats and resources. These multiple representations must be clearly integrated so that they do not cause additional cognitive load on students during the learning process.
- *Inter-example features*: Using more than one worked-example allows students to identify similarities and differences among them, improving the learning experiences.
- *Interacting with the learning environment*: Students should be encouraged to engage studying the worked examples. A possible engagement process is the use of the self-explanation effect.

Several authors have explored the characteristics of effective self-explanations[18-20]. Chi and colleagues[18] were pioneers in exploring self-explanations for worked examples. They argued that it is possible to learn from a single worked example as long as there is a thorough self-explanation process. Findings from Chi et al. [18] differentiated good and poor explainers based on the number and nature of their explanations. Good explainers produced more explanations and more monitoring statements compared to poor explainers, but also relied less frequently on the examples when solving problems. The results also suggested that, in order to show understanding of the example, four elements must be present in an explanation: (1) the conditions of the application of actions; (2) the consequences of actions; (3) the relationship of actions to goals; and (4) the relationship of goals and actions to natural laws or other principles.

One of the strategies that have been explored to elicit self-explanations in computer programming environments is the use of in-code comments[21]. However, this technique has not been evaluated in the context of computational science education. This paper explores the characteristics of self-explanations as in-code comments in the context of computational science education.

Methods

Participants and Procedures

Twenty-seven students enrolled in a freshmen level course called Computation and Programming for Materials Scientists and Engineers (CPMSE) participated in this study. The course was introduced into the Materials Science program at large Mid-Atlantic university as part of a curricular innovation intended to prepare future scientists and engineers with the knowledge and skills to use and develop computational tools and methods within their discipline[4].

Students were exposed to an inverted classroom approach in which they were required to watch lecture videos before the class. During class time, students worked on programming exercises that included at least one worked example using multiple forms of representation. These in-class activities comprised 16 sessions of the semester. Starting on in-class activity

number nine, students were given the possibility of having 10-point extra credit (for a 1000-point course) for writing comments within the MATLAB® code describing what the worked-example was doing at each code line.

This paper focuses on qualitatively analyzing student self-explanations for in-class activity nine. The identified characteristics of the self-explanations will be compared to student performance on project three, the one that followed the in-class activity nine.

Materials

The worked-example for the in-class activity nine consisted of a two-dimensional bouncing simulator for an imperfectly elastic ball (see the sample code in Figure 1). The worked-example included the problem statement, a description of what was being asked to do, an incorrect version of the MATLAB ® code, and a corrected version of it. The purpose was to expose students to debugging procedures in MATLAB®. The example also included a video explanation of the debugging procedures and how to fix different bugs contained in the incorrect code. All these resources can be accessed at http://cvieira77.wix.com/cpmse-ex4-act5

Project three was in the area of molecular dynamics and consisted of three parts. In the first part, students were introduced to the concepts of molecular dynamics. Two commented examples on text file reading and curve fitting were presented to them. The second part asked students to write a program to read a file including x, y, and z coordinates of N diffusing ethanol molecules as a function of time. Their program should calculate the diffusion coefficient and plot the fitted curve. The third part asked them to write a program that simulated a one-dimensional random walk.

Data Collection

Two main sources of data collection are considered for this study. The first source is the set of in-code comments students wrote as self-explanation of the worked-example. Two sample commented codes submitted by the students are depicted in Figure 1. Note that the differences in these two students' approach to self-explaining are not limited to the extension of the comments. While student A did not describe the purpose of the function and each of the parameters, student B did. Also, student A described the code in terms of the data structures (e.g. matrix, vector) and operations between them, while student B consistently used science concepts (e.g. *"…the overall velocity to decrease if the ball is not perfectly elastic…"*) to describe the code. Finally, note that student B described the assumptions and conditions under which the program was designed.

```
function bounceserr(N, width, height, loss)

 pos = zeros(2,N+1);% sets pos to a matrix of zeros with 2 rows and N+1 columns
 pos(:,1)=rand(2,1).*[width; height]; %sets the vector pos(:,1) equal
 % to the matrix multiplication
 %of a randomely generated 2 by 1 matrix and a 2 by 1 matrix using the
 %values of the inputs width and height
 dir = rand(2,1);% first assigns the variable dir to a randomly generated
 dir=dir/norm(dir);%2 by 1 matrix then redefines it as the variable divided by
 %the matrix norm of dir
 %for the follwing condition from n=2 to N+1 the matrix [pos(:n),dir]
 %equals the vector of new pos which consists of pos(:,n-1) dir, loss,width
 %height
 for n=2:N+1
 [pos(:,n),dir]=newpos(pos(:,n-1),dir, loss, width, height);
 end
```

(a)

```
function bounces(N, width, height, loss)
% bounces plots the trajectory of a ball in a 2 dimensional box.
% N is the number of bounces.
% width is the width of the box.
% height is the height of the box.
% loss is the loss coeffecient, where 1 is totally elastic and 0 is totally
% inelastic.
% The ball is assumed to travel in a straight line
% When the ball hits a wall, the component of velocity parallel to the wall
% does not change, but the component of the velocity perpendicular to the
% wall is multiplied by the opposite of loss (causing the overall velocity
% to decrease if the ball is not perfecty elastic and causing the
% trajectory to switch directions, giving the "bounce" effect)

%     Creates a position matrix where each column is a new (x,y) point,
%     where the first row is the x coordinate and the second row is the y
%     coordinate.
    pos = zeros(2,N+1);
%     Assigns a random starting location within the box. rand(2,1) will
%     give a number between 0 and 1 and multiplying by [width; height]
%     ensures that the position will be within the box. This position is
%     assigned to the first column in pos.
    pos(:,1)=rand(2,1).*[width; height];
%     Assigns a random direction for the ball to be going at first.
    dir = rand(2,1);
%     Makes sure that the direction is of unit size so that it does not
%     increase the velocity of the ball.
    dir=dir/norm(dir);

%     The first bounce starts at n = 2 and it will bounce until the loop
%     reaches N+1. The loop is in increments of 1
    for n=2:N+1
%         See the "newpos" function below for an explanation of why this
%         works
        [pos(:,n),dir]=newpos(pos(:,n-1),dir, loss, width, height);
    end
```

(b)

Figure 1 – Sample comments within the worked-example for two different students

The second data source is the students' score for project three. This project was the one that students completed after in-class activity number nine. The project was organized following the modeling and simulation process[22], and was scored using an assessment rubric aligned with this process (see the rubric in Appendix A). Although this project is not a direct assessment of student understanding of the worked-example, it is an indicator of student learning process. We will employ this project score to characterize students' self-explanations. The limitation of this approach is that we will not be able to identify causality between self-explanations and performance.

Data Analysis

The MATLAB® code of the worked-example was divided into 15 different sections in which the student could have written comments. These sections could involve one or more lines of code according to their purpose. For example, the following code was considered a single section intended to make a decision using an if-else clause.

```
if(dir(1)<0)
    xwall=0;
else
    xwall=w;
end
```

A coding scheme was designed based on previous research on self-explanations[18-20]. The qualitative data set was first analyzed by one researcher who refined the coding scheme on the basis of the data. A second researcher independently coded 20% of the data, and shared the findings with the first researcher. The disagreements were negotiated between the two researchers, and the coding scheme was refined once more.

We started the coding with the four elements suggested by Chi et al.[18] :

- The conditions of the application of actions;
- The consequences of actions;
- The relationship of actions to goals; and
- The relationship of goals and actions to natural laws or other principles.

Then, ten different codes were identified from preliminary analysis of the data (see Table 2). Note that the first five codes (i.e., CAA, COA, RAG, RAL, and DC) correspond to understanding of the example, and can appear simultaneously in a single section of the MATLAB code. The other six codes (i.e., NE, RS, DE, IS, RP) focus on the type of comments students write, which are mutually exclusive, meaning that only one of them can represent a single section. Finally, a single section can be assigned codes from both groups.

The number of instances of each code from the coding scheme within each student file is counted. These are compared between the high performers and the low performers in the project three.

Table 2 – Coding scheme for students' in-code comments as self-explanations

# of instances	***Code***	***Description***	***Example***
Zero, one, or multiple	***Identifying the conditions of application of actions (CAA)***	*The student identifies the **parameters** or **conditions** under which an instruction is executed.*	*for the **follwing condition from n=2 to N+1** the matrix [pos(:n),dir] equals the vector of new pos which consists of pos(:,n-1) dir, loss,width height*
	Describing the consequences of Actions (COA)	*The student briefly describes **what** an instruction is going to do when executed.*	*% bounces **plots the trajectory of a ball** in a 2 dimensional box*
	Describing the relationship between actions and goals (RAG)	*The student describes the reasons **why** an instruction or operation is carried out.*	*% When the ball hits a wall, **the component of velocity parallel to the wall does not change, but the component of the velocity perpendicular to the wall is multiplied by the opposite of loss (causing the overall velocity to decrease** if the ball is not perfecty elastic and causing the trajectory to switch directions, giving the "bounce" effect)*
	Connecting to disciplinary laws or principles* (*RAL)	*The student makes connection with the **disciplinary concepts** to describe the instructions.*	*This will take in values that will be used to simulate the bouncing of a **perfectly inelastic ball***
	Describing the code instead of the program (DC)	*The student describes the programming code in terms of data structures and operations.*	*sets the vector pos(:,1) equal to the matrix multiplication of a randomely generated 2 by 1 matrix and a 2 by 1 matrix using the values of the inputs width and height*
One	***No explanation (NE)***	*The student does **not include any explanation**.*	***N/A***
	Restatement (RS)	*The student writes **exactly what the code says** without depicting understanding of it.*	*This will take in values that will be used to simulate the bouncing of a perfectly inelastic ball*
	Description (DE)	*The student actually **describes** what the code is doing.*	*bounceserr is a function that will simulate a ball of choosen elactisity bouncing around the two dimensional plane of a graph off points on the x and y axis… The ball travels in a straight line and the velocity parallel to the wall doesnt change but the perpendicular velocity is multiplied by -loss.*
	Incorrect Statement (IS)	*The student provides an **incorrect description** of the instruction.*	*Makes sure that the direction is of unit size so that it does not increase the velocity of the ball.*
	Repeating (RP)	*The student identifies that the section is **similar** to a previous one with a change of parameters.*	*The next set of code lines **does the same thing as the one above, except this time it is for the y direction not the x**, hence why we are checking the y component of dir rather than the x.*

Results

Fourteen students submitted their self-explanations for extra-credit during the in-class activity number nine. Two of these students were not included in the qualitative analysis because they copied the comments directly from the video-explanation. Hence, their self-explanation approaches cannot be used to identify the self-explanation effect. For the remaining 12 students, the number of commented sections, the project score, and the number of instances for each category from the coding scheme within the students' self-explanations are presented in Table 3. Note that the students were given a pseudonym to protect their identity. Only Kerry reported having taken previous courses in programming.

Table 3 – Results of the qualitative analysis of student self-explanations

Student	Code									# Commented Sections	Project 3 Score
	CAA	COA	DC	DE	IS	RAG	RAL	RS	RP		
Devyn	4	11	0	10	0	0	0	0	0	13	74
Jody	5	11	1	10	0	8	7	1	0	12	75
Emerson	6	11	0	12	2	6	4	0	0	14	81
Infant	4	14	0	13	0	6	8	0	0	15	82
Santana	2	15	0	13	0	2	6	0	1	15	87
Jessie	12	14	0	14	0	11	11	0	1	15	88
Justice	4	8	13	0	1	0	0	11	0	14	90
Jaylin	5	12	1	14	0	4	9	1	0	15	92
Landry	9	11	1	14	0	8	9	1	0	15	93
Harley	11	12	8	6	1	2	4	1	0	15	94
Kerry	8	13	0	13	0	3	6	2	0	15	97
Riley	14	14	0	15	0	13	11	0	1	15	98

The students were divided as low performers (white) and high performers (green) using the mean score as the criterion (87.58). The mean score for students that did not complete the extra credit assignment was 86.89. On average, the high performers identified more conditions of application of actions (CAA), more relationship action-goals (RAG) and more connections to laws or principles (RAL). But also described more the code (DC). Both the low and the high performers presented incorrect statements (IS) as part of their self-explanations. Figure 2 depicts a graphical comparison of the average number of instances for each of the categories within the students' self-explanations.

Most of the students commented all the 15 sections of the worked-example, but high performers where more extensive and thorough, given the number of instances identified during the qualitative analysis. Justice, a high performer, is an interesting case because she did not provide any relationship between actions and goals (RAG), or any connections to laws or

principles (RAL). These two elements are more common (on average) for high performers. Instead, she focused on describing the code and not the problem itself.

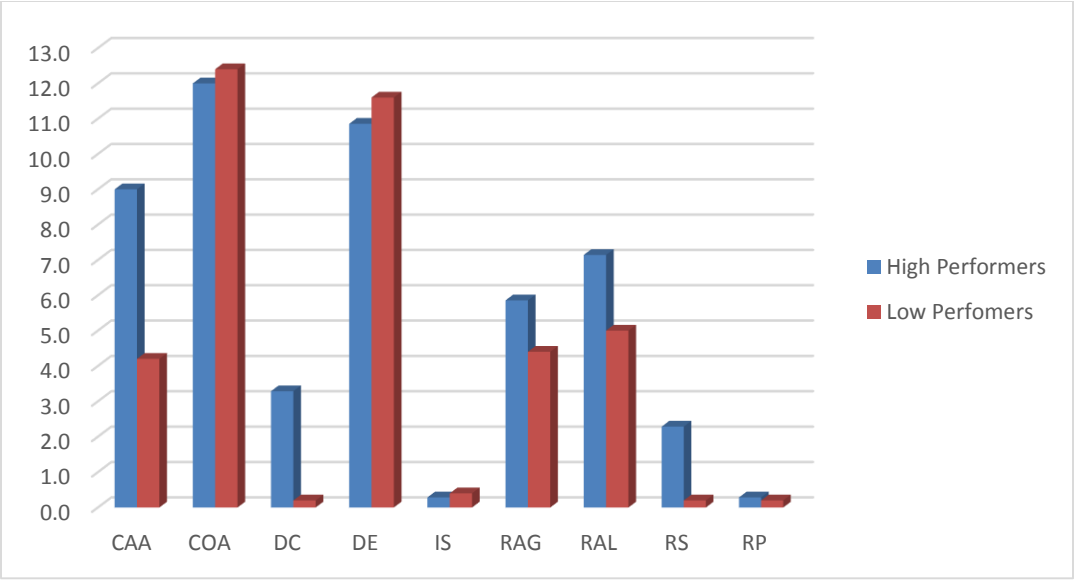

Figure 2 – Comparison between the average number of instances of each code from the high- and the low performers within student self-explanations

Discussion

What are the characteristics of student self-explanations of programming worked examples involving a disciplinary context?

Twelve students completed the self-explanation during in-class activity number nine for extra credit. Of these 12 students, eight commented all 15 sections, two students commented 14 sections, one commented 13, and one commented 12 sections. All but two of the students showed at least one instance of all the four elements that need to be present to depict understanding[18]: (1) the conditions of the application of actions (CAA); (2) the consequences of actions (COA); (3) the relationship of actions to goals (RAG); and (4) the relationship of goals and actions to natural laws or other principles (RAL). Five students wrote at least one comment in terms of the MATLAB® code (DC), and three students wrote incorrect statements (IS). Six students used restatements (RS) that did not depict an understanding of the purpose of the instruction, and only three students identified that one section was similar to a previous one with a change of parameters. Overall, the differences in student approaches suggest that we can identify effective in-code commenting practices in order to characterize the self-explanation effect [8, 14].

How are students' self-explanation characteristics related to their performance in a computational project involving a disciplinary context?

The high performers in project three differed from the low performers in four specific categories; particularly in identifying conditions of application of actions (CAA), relationship action-goals (RAG), connections to laws or principles (RAL), and describing the code (DC). Three of these categories (i.e., CAA, RAG, and RAL) are part of the four elements under which a student depicts understanding according to Chi and collaborators[18]. The difference in the average number of RAL instances also agrees with the finding by Renkl[19] who described the good explainers as 'principle-based' (i.e., those who used laws or principles in their self-explanations). The fourth category identified (DC) contradicts Pirolli and Recker's[20] findings who suggested that poor performers would focus more on superficial characteristics compared to high performers. In this study, Justice and Harley focused on describing the code, which can be seen as the superficial part of the worked example. Nevertheless, the performance measure that we employed is not a direct assessment of student understanding of the worked-example. As a consequence, further research is necessary to investigate the relationship between students' explanations and their understanding of the worked-examples.

Conclusions

We explored the use of student in-code comments as a self-explanation strategy in a computational science education context. Students wrote comments for a large number of sections in the code. Some of these explanations made connections with the disciplinary knowledge, while some others focused on describing the code. Other differences were identifiable from the student in-code comments. For instance, some students described the purpose of the functions, the parameters, and the assumptions under which the program works. Some other students preferred to focus on superficial features of the code such as the data structures.

Students with high and low performance in the project three differed in their self-explanation approaches. The project is not an assessment of student understanding of the worked example, but provides a measure to characterize the student's self-explanation process. Although in-code comments as self-explanations are limited to what students write, this approach offer additional advantages compared to think-aloud protocols to elicit self-explanations. First, teachers can implement in-code comments, both as a classroom activity and as a homework assignment. Second, assessing explanations from written comments is less time-consuming than qualitatively analyzing think-aloud transcripts. The drawbacks of this approach compared to think-aloud protocols are that we are unable to identify monitoring statements or additional reasoning students make during the self-explanation process.

Limitations and future work

The main limitation of this study is that students wrote the self-explanations in a voluntary basis. Students were given extra credit for completing them, and therefore, they may not represent the whole spectrum of students in the classroom. Most likely, students who completed the self-explanations either did it because they really needed the extra credit, or

because commenting on the code did not require significant effort on their part given their level of comfort with programming.

A second limitation involves the use of the project three score as a measure of performance to compare the different self-explanations. Project three involved more than what could have been learned from the worked-example. As a consequence, this study does not allow us to identify a cause-effect relationship between self-explanations and performance.

Nevertheless, the possibility of identifying different self-explanation approaches using in-code comments encourages us to follow this path of research for computational science education. The findings from this study and other in-progress research processes allow us to persuade the faculty members to having mandatory self-explanations within the classroom, as well as direct assessment instruments of student understanding about the examples. Future steps include the use of a larger and more representative sample, the analysis of self-explanations within certain section types, and the use of think-aloud protocols to remove all the external confounding variables.

Acknowledgements

This research was supported in part by the U.S. National Science Foundation under the awards #EEC1329262 and #EEC1449238.

References

1 WTEC.International assessment of research and development in simulation-based engineering and science. World Technology Evaluation Center, Inc., Baltimore, Maryland. (2009)

2 NSF. National Science Foundation Advisory Committee for Cyberinfrastructure Task Force on Grand Challenges Final Report, (March). (2011)

3 PITAC. Computational science: ensuring America's competitiveness. President's Information Technology Advisory Committee (PITAC), vol. 27 (2005)

4 Authors, 2013..

5 Magana, A.J., Vieira, C., Polo, F.G. Yan, J. and Sun, X.. An Exploratory Survey on the Use of Computation in Undergraduate Engineering Education. Frontiers in Education Conference. Oklahoma City, OK. October 23-26. (2013)

6 Rogalski, J., & Samurc¸ay, R. 1990. Acquisition of programming knowledge and skills. In J.M. Hoc, T.R.G. Green, R. Samurc¸ay, & D.J. Gillmore (Eds.), Psychology of programming (pp. 157–174). London: Academic Press

7 du Boulay, B. 1989. Some difficulties of learning to program. In E. Soloway & J.C. Spohrer (Eds.), (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum.

8 Sweller, J., Ayres, P., and Kalyuga, S. 2011. Cognitive Load Theory, Explorations in the Learning Sciences, Instructional Systems and Performance Technologies, doi: 10.1007/978-1-4419-8126-4_5, Springer Science+Business Media, LLC

9 Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. 2008. Programming by Choice: Urban Youth Learning Programming with Scratch. SIGCSE'08.

10 Vieira, C., Magana, A.J., Roy, A., Falk, L.M. & Reese, J.M. (2015). Exploring Undergraduate Students' Computational Literacy in the Context of Problem Solving. In Proceedings of the 122th ASEE Annual Conference and Exposition. Seattle, Washington. June 2015.

11 Shaikh U.A.S., Magana A.J., Vieira, C., & Garcia, R. E. 2015. An exploratory study of the role of modeling and simulation in supporting or hindering engineering students' problem solving skills. In Proceedings of the 2015 ASEE Annual Conference and Exposition. Seattle, Washington. June 2015.

12 Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L. 2011. Computational thinking for youth in practice. ACM Inroads, 2, 32–37

13 Anderson, J. R., Fincham, J. M., & Douglass, S. 1997. The role of examples and rules in the acquisition of a cognitive skill. Journal of Experimental Psychology: Learning, Memory, and Cognition, 23(4), 932-945.

14 Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. 1989. Self-explanations: How students study and use examples in learning to solve problems. Cognitive Science, 13(2), 145–182.

15 Kuhn, D. & Katz, J. 2009. Are self-explanations always beneficial? Journal of Experimental Child Psychology 10. 386–394

16 Lombrozo, T. 2006. The structure and function of explanations. Trends in Cognitive Sciences, 10, 464–470.

17 Aleven, V. & Koedinger, K.R. 2002. An effective metacognitive strategy: learning by doing and explaining with a computer-based cognitive tutor. Cognitive Science 26, 147–179

18 Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. 1989. Self-explanations: How students study and use examples in learning to solve problems. Cognitive Science, 13(2), 145–182.

19 Renkl, A. (1997). Learning from worked-out examples: A study on individual differences. Cognitive Science, 21(1), 1–29.

20 Pirolli, P., & Recker, M. (1994). Learning strategies and transfer in the domain of programming. Cognition and Instruction, 12, 235-275

21 Vieira, C., Yan, J., & Magana, A.J. (2015). Exploring Design Characteristics of Worked Examples to Support Programming and Algorithm Design. Journal of Computational Science Education, 6(1).
22 Shiflet, A. B., & Shiflet, G. W. (2014). Introduction to computational science: modeling and simulation for the sciences. Princeton, New Jersey: Princeton University Press.

Appendix A

Due to space limitations, a reduced rubric describing only the lowest and the highest score is presented here.

Criterion	Description	Poor (0-2)	…	Excellent (9-10)
Analyze and formulate the problem (10%)	Evaluates the student's plan for completing the project. Student instructions: Summarize the nature of the algorithm briefly, identifying the most relevant information from the project description. Articulate a well thought-out strategy for designing, coding, testing and debugging your work.	- No strategy is articulated for the design, coding, testing or debugging.	…	- All four areas (designing, coding, testing, debugging) are addressed clearly in the context of the project. - The summary references the project description and identifies relevant aspects of the project. - The strategy is articulated clearly and is logical and well thought-out.
Solve the problem (40%)	*Coding style (10%)* Measures the extent to which the code is presented in a manner that is clearly readable by others. Is the code indented, commented and are variable and function names chosen to enhance readability? Does the code appropriately deploy language capabilities to avoid redundant structures, global variables and unnecessarily lengthy blocks of code?	- Code is entirely uncommented. - Global variables are used without justification due to exceptional circumstances. - Code is not differentiated into functions or m-files; i.e. spaghetti code.	…	- Code is well commented. - Code is properly indented and variable and function names are well chosen. - Code is well structured.
	Program execution (30%) Evaluates the extent to which the program functions in a way that conforms to specifications. Does the program execute? Is the input and output of the expected form?	- Program does not run at all.	…	- Program is free of syntax errors that impede execution. - Program takes the expected input parameters and returns the expected output as required in the specification in all respects.
Verify the solution (30%)	Evaluates the degree to which the solution satisfies the specification. Is the solution accurate and robust? Does it conform to the problem specifications regarding format, order and presentation?	- The solution produces wholly incorrect output under all of the tests run.	…	- The solution produces correct output in all cases with only minor exceptions. - All output meets specifications regarding format, order and presentation.
Interpret and report the solution (20%)	Evaluates whether the student can use the solution to approach a disciplinary problem. Can the student use their code to address the disciplinary issue or to solve a related problem?	- No solution provided.	…	- A solution is provided that is correct, clear and well documented.