Shih, J. L. et al. (Eds.) (2019). Proceedings of the 27th International Conference on Computers in Education. Taiwan: Asia-Pacific Society for Computers in Education

A Crowd-Programming Approach for Computational Thinking Education

Christopher KRIZANOVIC, Nguyen-Thinh LE & Niels PINKWART
Department of Computer Science, Humboldt-University Berlin, Germany
Nguyen-thinh.le@hu-berlin.de

Abstract: This paper proposes a crowd-programming approach for computational thinking education. An evaluation study was conducted with 61 high school students to investigate four hypotheses: 1) Learners using the computer-supported crowd-programming system require less time for problem-solving, 2) they need less attempts for correct solutions, 3) they tend not to request bottom-out hints for programming problems, and 4) they can solve more programming problems within 70 minutes than in other conditions.

Keywords: crowd-programming, peer-review, pair-programming, computational thinking, didactics for computer science education

1. Introduction

Problem solving has been considered one of the key elements of computational thinking (Wing, 2006). The Computer Science Standard K12 framework developed by the Computer Science Teacher Association (CSTA) characterizes computational thinking as a problem solving process that applies selected strategies (K12CS). Dagiene and colleagues (Dagiene et al., 2016) developed a two-dimensional categorization system for the framework BEBRAS that takes account of computational thinking skills as well as concepts in Computer Science. The BEBRAS framework is also based on problem solving. Unlike the frameworks for computational thinking relying on Computer Science education (CSTA K12 and BEBRAS), the PISA framework for Computational Thinking is supposed to be a part of Mathematics (Lorenceau et al., 2019). This framework considers computational thinking skills as part of the problem-solving process.

In order to support learners in solving programming problems, and thus, in enhancing their computational thinking skills, software development strategies could be involved. In this paper, we present a crowd-programming approach that is based on two software development strategies: peer-review and pair-programming. The crowd-programming approach is deployed in a computer-supported learning system, which is intended to support learners in improving their computational thinking skills.

2. The Crowd-Programming Approach

Peer-review is a software development process that is monitored and reviewed by one or several persons. That is, a software solution of a developer is examined by one or several other developers. It is typically applied to assure the quality of software development and is considered as an indispensably useful software development strategy (Rigby et al, 2008). Adopting the criteria for effective software development strategies (Cockburn & William, 2000), we could derive that the peer-review approach contributes only to solve problems that occur during the review process. This strategy does not involve directly in the problem solving process, i.e. a solution must be developed by the developer. In the context of learning, adopting this approach, if a learner is stuck, he/she would not be able to find a solution for a programming problem. Regarding the criterion for detecting errors in the early stage, this approach must suffer, because the review process takes place at a later time point. Regarding the criterion "learning", reading peer-review is the opportunity for enhancing their programming skills and also, the task of writing a review for peers' solutions could be another opportunity to learn new solution strategies.

A special variant of peer-review is the in-flow peer-review that requires the review process to take place *during* the software development process (Clarke et al., 2014). Considering the criteria for an effective software development strategy (Cockburn & William, 2000), the traditional peer-review approach and the in-flow peer review overlap at several points. Of course, the in-flow peer review approach

contributes to problem solving at the early stage of the software development process (e.g., design an algorithm, specification of procedures). The in-flow peer-review approach enables the detection of errors at the early stage.

Pair-programming is a software development strategy that requires two programmers to solve a programming problem together. Both persons discuss and solve the same problem constantly and develop a solution for the problem collaboratively. One person is responsible for coding, another one acts as observer, who permanently checks the code and gives feedback. The pair-programming approach has the advantage that both programmers may make exploit knowledge and programming experience from each other. Thus, problems could be solved faster. Cockburn and William (2000) reported that pair-programming contributes to detecting errors at an early stage of software development. As both developers can contribute their knowledge and programming experience in problem solving, both can benefit from each other.

Although the three presented software development strategies are established and widely deployed in the software development branch, they may suffer several disadvantages for learning purposes. The disadvantage of the peer-review approach is that through the late review task, learners do not have the chance to receive hints for problem-solving at early stages. This disadvantage can be solved by the in-flow peer review, because the review tasks take place during the whole period of problem-solving process. The in-flow peer review approach may also have disadvantages. First, learners, who do not have an interest in solving programming tasks by themselves, could misuse the review task in order to copy solutions from peers. A second disadvantage could be caused by imprecise or incorrect reviews, which could mislead the author(s) of a solution. Also the pair-programming approach may not work advantageous as we wish. If a pair consists of an expert and a novice programmer, in this constellation, the expert programmer would have to manage the software development task alone and the novice programmer would not learn anything from the collaborative work.

In order to circumvent the disadvantages of the software development strategies discussed above, this paper proposes the so-called crowd-programming approach. This approach is intended to enable the learner to access hints from peers. Hints are not limited to one single peer, but are created as a collective pool by all learners. Thus, the learner can use the pool of hints and choose the appropriate ones to carry on to solve a specific problem. Learners do not have to wait to the end of a review process (like for the case of peer-review), hints can be provided at real-time. The computer-supported system applying the crowd-programming approach can collect hints (that have been given in the past) regarding a specific problem and generate a review for the learner.

After consulting a review (in form of a list of hints) for the specific problem and once the learner is able to solve that problem, he/she is required to evaluate those hints. The evaluation of hints is referred to as meta-review in the crowd-programming approach. This mechanism helps the system to filter out reviews that have been submitted randomly and are not relevant to the specific problem and to generate only reviews with highest meta-review score. The mechanism of optimizing the quality of review to be provided is intended to help the learner to solve the specific problem without being mislead by incorrect/irrelevant reviews. This approach makes use of the collective intelligence from the crowd, thus, the approach being presented in this paper is called crowd-programming approach. The research question to be investigated in this paper is: how effective can the crowd-programming approach support learners as they solve programming problems?

3. SUPELS: A Crowd-programming Based System for Computational Thinking Education

3.1 Requirements

The aim of this system is to apply the crowd-programming approach to support students learn the foundations of programming and to improve their competency in solving programming problems. Thus, the system will have to provide learners with a programming course that consists of lessons. Each lesson is composed of an introduction text for a new programming construct and exercise assignments. The task of the learners is to read the introduction text and to solve those exercise assignments. As a programming language to be learned, JavaScript was suggested, because this programming language is not commonly taught at schools. This is to avoid possible biases in the experiment study to be conducted. If the study subjects had learned the programming language, they would solve the exercises provided by the system quickly. The topics of the ten lessons are: syntax of JavaScript, comment, variable, variable modification, function, data types and Boolean, logical operators and comparatives

(for number), logical operators and comparatives (for Boolean), If-statement, For-statement. The introduction texts and exercise texts should be expressed in an understandable way for high school students. Thus, the composition of ten lessons above was consulted from the books "JavaScript for Kids" (Morgan, 2014) and "JavaScript for Dummies" (Harris, 2012).

3.2 The Problem-solving Scenario

Introducing New Lessons: The first user interface is used to display the introduction text and the corresponding exercise assignment. The introduction text can be folded up after reading in order to focus only on the exercise assignment and can be folded out according to the need of the learner. This adaptive presentation is an important feature required for educational systems to adapt to individual memory capacity (Lestari et al., 2017).

Solving Exercises: The second user interface is provided to solve an exercise assignment. To solve an exercise, a text editor is required and several buttons for different functions (e.g., test code, reset code, show hint, show bottom-out hint) are required. Depending on the lesson and the exercise, the text editor is empty or display with code that requires the learner to extend or to adapt. The function "test code" runs the code against a test bed and compares the submitted code with correct solutions (that have been edited by the exercise author or that have been submitted by the learner and automatically validated by the system as correct). If the submitted code is evaluated as incorrect, the learner is asked to consult peer-review using the function "show hint". After three unsuccessful attempts, the learner will have the possibility to use the function "show bottom-out hint", which shows the correct solution for the exercise assignment.

Displaying Hints: The third user interface (Figure 1) is used to display hints. The displayed hint can be meta-reviewed by the learner as positive (thumb up) or negative (thumb down). If one hint is not enough, the learner can request the next hint (because there is a collective pool of hints for a specific problem). In order to select appropriate hints from the collective pool, the programming code submitted by the learner needs to be analyzed. The code analysis is based on the comparison between the learner's code and the anticipated correct solution and all differences are identified. The differences indicate what the learner has to modify on his/her code in order to reach a correct solution. The coding state, at which the learner asks for hints, is the specific problem state of the learner. The system looks for corresponding reference coding state in the database and choose associated hints.

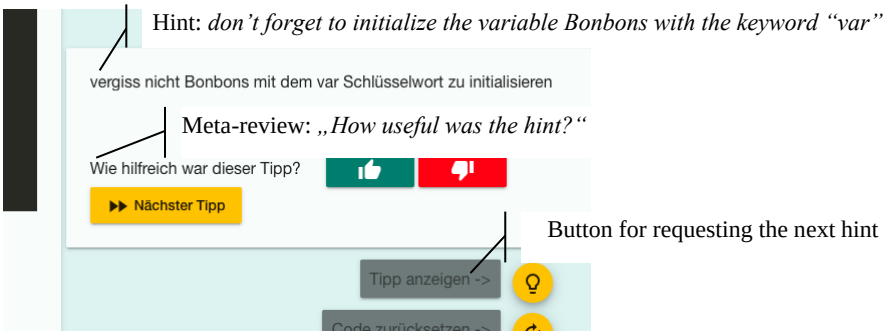

Figure 1. The user interface for displaying requested hints

In order to select the best hints in the collective pool for a specific coding state, the distance between the learner's code and the reference coding state is computed using the Levenshtein algorithm, because algorithm is commonly used as a metric for string matching (Navarro, 2001). The Levenshtein distance is computed by the number of deletion, addition, and replacement operations that are required to transform the learner's code to the reference coding state (or vice versa). The algorithm for choosing the best hints follows. The Levenshtein distance between the learner's code and the referenced code of a feedback is computed. Then, the distance is divided by the length of the referenced code. If the division value is 1, the Levenshtein distance and the length of the referenced code are equal. The best hint is chosen, if it has not been shown and its division value is highest (i.e., closed to value 1). In addition to Levenshtein distance, meta-review values for hints are also considered. If two hints have a difference of the division value within the threshold of 0.2, the hint with higher meta-review value will be chosen. General feedback (e.g., regarding coding style) that does not have a referenced code will also be selected if it has not been displayed yet and has a highest meta-review value.

Inputting hints: For inputting hints, the fourth user interface (Figure 2) is required. Those learners who have completed an exercise assignment successfully, are asked to input hints and suggestions for helping peers, who will have to go through the lessons later. The procedure for inputting hints is following. When the learner receives the information that his/her solution is correct, he/she is asked to leave hints which may be helpful for peers later. Then, the learner should think about which code line(s) were difficult for him/her and would be problematic for peers. The learner should mark those code lines (which are referred to as reference coding state) and give hint. A learner can input several hints and different reference coding states.

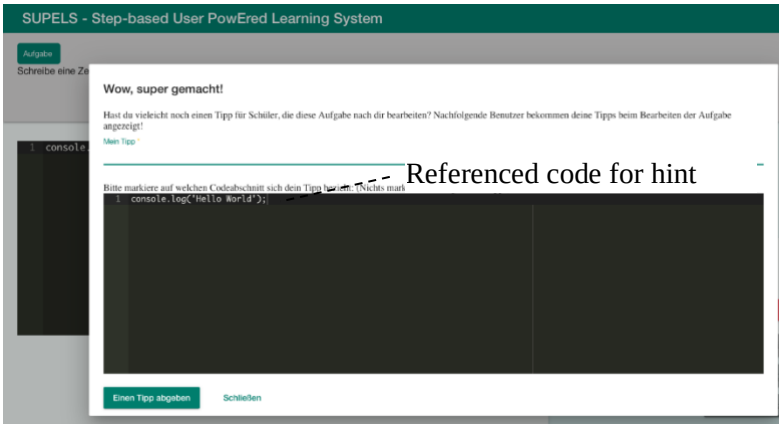

Figure 2. The user interface for inputting hint

4. Evaluation

4.1 Goal

The goal of the study addressed in this paper is to investigate the research question: how effective can the crowd-programming approach support learners to solve programming problems? Since the proposed crowd-programming approach is intended to develop computational thinking, the target group of the developed system includes high school students, who have just started learning Computer Science at schools.

4.2 Design and Participants

The study consists of three phases: 1) a pre-test (10 minutes), 2) the experiment phase (70 minutes), and 3) a post-test (10 minutes). The pre-test is intended to measure knowledge and programming skills of students. The post-test serves to measure the improvement of problem-solving performance after using the learning system. The pre-test and post-test consist of test items for programming constructs that are addressed in the ten lessons provided by the learning system.

A post-test was given to the participants of all conditions after completion of the experiment session. Pre- and post-test were made comparable by a counter-balanced design of the test items. Specifically, two test versions were developed: Test A and Test B. Test A was assigned as pre-test and Test B as post-test to 50% of the participants of each condition, and the rest of the control group had Test B as pre-test and Test A as post-test. The whole process including pre-test, experiment session, post-test, and questionnaire was limited to 90 minutes. Each test can be achieved with a maximum of 21 points. Learning gains are defined as the difference between the score of post-test and pre-test.

Prior to the pre-test, information about the learners were collected. This includes information about the experience of the learners, experience with JavaScript. In case the learner did not have experience in programming, a further question was asked about why he/she has not learned programming. In addition to the post-test, students get a questionnaire for evaluating each learning condition (i.e., using the computer-supported crowd-programming system, pair-programming setting and the control group without help). Each question was rated on the scale between 1 (very bad) and 5 (very good).

During the experiment period, numbers of submission attempts, time required for solving each exercise assignment, requests for bottom-out hints, frequency of requests for feedback were recorded. Student solutions of tests and of experiment exercises were collected to be used for analysis. In addition, students' responses to the questionnaire were used as subjective data.

We contacted high schools in Berlin and could acquire 64 students from grade 8 to 12. Since 3 Pre/Post-tests were missing, these data entries could not be considered. At the end, we had a sample size of 61 participants that were distributed in three conditions whereas more participants were assigned to the

experiment condition than in other conditions because the SUPELS system should be evaluated not only quantitatively but also qualitatively: Condition 1: 30 participants in the experiment group suing the SUPELS system; Condition 2: 20 participants in the condition with pair-programming using a software development environment; Condition 3: 11 participants in the control group (using introduction texts from course books for solving JavaScript programming exercises).

4.3 Results

Hypothesis 1a: Learners in Condition 1 requires less time for problem-solving than in other conditions.
Hypothesis 1b: Learners Condition 1 needs less solution submission attempts than in other conditions.
Hypothesis 1c: Learners in Condition 1 less request bottom-out hints than other conditions.
Hypothesis 1d: Learners in Condition 1 achieves higher number of lessons in a specific time period (70 minutes) than in other conditions.

In order to test the hypotheses 1a, 1b, and 1d, the one-factorial variance analysis ANOVA can be used to compare the difference between two independent groups, because each group has an independent variable. Considering Hypothesis 1a, the dependent variable is time required for solving exercise assignments. Applying the Shapiro-Wilk-Method, time required for solving exercises with the Gaussian distribution was not normally distributed (α=0.05) for all three conditions. However, research studies showed that one-factorial ANOVA is robust for violation of normal distribution (Lix et al., 1996; Salkin, 2010) and the violation of normal distribution is not critical if the data sample size is over 25. The Levene-Test shows that homogeneity of variance is present (p=0.092). Thus, the ANOVA analysis is valid. For testing Hypothesis 1a, there were 274 data entries of crowd-programming, 95 of pair-programming, and 109 of control group. Each data entry represents time required for one lesson (reading an introduction text and solve an exercise successfully) by one subject, and thus is independent. Table 1 shows that the crowd-programming group required more time (m=169.3; sd=159) than other groups, but the difference between the groups is statistically not significant (F(2, 475)=0.88; p=0.42) at a significance level of 0.05. Thus, Hypothesis 1a can be rejected.

Table 1
Results for Hypotheses 1a and 1b

	Condition 1	Condition 2	Condition 3	F	p
Time for problem-solving (seconds)	169.3 (sd=159)	161.6 (sd=145.2)	146.5 (sd=147)	0.88	0.42
# submission attempts	2.66 (sd=5.34)	3.86 (sd=3.12)	2.87 (sd=2.48)	2.60	0.08
# completed lessons	9.13 (sd=2)	9.5 (sd=1.08)	9.9 (sd=0.3)	2.58	0.10

Regarding Hypothesis 1b, there were also 274 data entries of crowd-programming, 95 of pair-programming, and 109 of control group. Each data entry represents the number of submission attempts required by each subject to achieve the correct solution for each exercise assignment, and thus is independent. Table 1 shows that the pair-programming group (m=3.86; sd=3.12) needed more submission attempts than other groups, but the difference between the conditions is statistically not significant (F(2, 475)=2.6; p=0.075) at a significance level of 0.05. Hypothesis 1b can be rejected.

Table 2
Results for Hypothesis 1c

B (coefficient)	SD	Df	p	Exp(B)
-0.63	0.19	1	0.001	0.53

For Hypothesis 1c, 274 data entries of crowd-programming, 95 of pair-programming, 109 of control group were collected. Each data entry represents whether a learner requested the bottom-out hint (i.e., correct solution provided by the system) or developed a correct solution for an exercise assignment by him-/herself. Thus, the data entries are independent. The probability of requesting a bottom-out hint is 0.49 (sd=0.5) for the crowd-programming group, 0.72 (sd=0.45) for the pair-programming group, and 0.58 (sd=0.5) for the control group, respectively. This statistical result shows that learners in the crowd-programming group requested bottom-out hint least frequently. In order to examine whether the difference between the groups is statistically significant, logistic regression is used for analysis. Since

the dependent variable only has the value 0 (learner has not requested the bottom-out hint) and value 1 (learners has requested the bottom-out hint), the assumption for binomial distribution is satisfied. Table 2 shows that the beta coefficient is negative (B = -0.63) with a standard deviation of 0.19 and Odds Ratio is less than 1 (Exp(B)=0.53). Thus, the probability that a learner of the crowd-programming group requesting bottom-out hints decreases by 63%. In addition, this probability is significant (p=0.001).
For testing Hypothesis 1d, the number of completed lessons is calculated, which ranges between 0 and 10. A completed lesson means that a learner has solved the associated exercise assignment successfully. For 10 lessons, there were 70 minutes. Table 1 shows that the control group could (m=9.9, sd=0.3) complete more lessons than the pair-programming group (m=9.13, sd=2.0) followed by the crowd-programming group (m=9.37, sd=1.62). Note, the standard deviation of the pair-programming group is relatively high. The result of WELCH test F(2, 20.54)=2.58 shows that the difference between the groups is statistically not significant (p=0.10) at a significance level of 0.05. Thus, Hypothesis 1d can be rejected.

5. Conclusions

This paper presents the crowd-programming approach to be deployed in a computer-supported learning system for computational thinking education. The crowd-programming approach exploits the advantageous characteristics of the software development strategies: in-flow peer-review and pair-programming. The current evaluation study investigates the research question whether the crowd-programming approach is an effective approach to improve problem-solving performance. The study shows that the crowd-programming based learning system did not help students reduce time for problem-solving, neither did learners of this condition require less solution submission attempts or achieved more lessons and exercise assignments than in other conditions. However, learners applying the crowd-programming approach requested fewer bottom-out hints than in other conditions. That means, crowd-programming learners tend rather to develop their own solutions than simply ask for correct solutions (bottom-out hints). The behavior of self-developing solutions would more help learners to develop problem-solving skills than merely requesting correct solutions. Authors of this paper are investigating other impacts of the crowd-programming approach (i.e., learning gains) and analyzing subjective attitudes of learners. These results are going to be published in near future.

References

Benos, D. J., Bashari, E., Chaves, J. M., Gaggar, A., Kapoor, N., LaFrance, M., ... Zotov, A. (2007). The ups and downs of peer review. *American Journal of Physiology - Advances in Physiology Education*, 31(2), 145-152.

Clarke, D., Clear, T., Fisler, K., Hauswirth, M., Krishnamurthi, S., Politz, J. G., Tirronen, V., & Wrigstad, T. (2014). In-Flow Peer Review. In *Proceedings of the Working Group Reports of the Innovation & Technology in Computer Science Education Conference*, ACM, 59-79.

Cockburn, A. & Williams, L. (2001). The costs and benefits of pair programming. In *Extreme programming examined*, G. Succi & M. Marchesi (Eds.). Addison-Wesley Longman Publishing Co., 223-243.

Dagiene, V., Sentence, S. & Stupurienė, G. (2016). Developing a fine-grained two-dimensional categorization system for educational tasks in informatics. Informatica 28(1):23-44.

Harris, A. (2012). JavaScript für Dummies. Publisher: Wiley-VCH Verlag.

Lestari, W., Nurjanah, D., Selviandro, N. (2017). Adaptive Presentation Based on Learning Style and Working Memory Capacity in Adaptive Learning System. In Proceedings of the 9th International Conference on Computer-Supported Education.

K12CS (2016). K–12 Computer Science Framework. Retrieved from https://k12cs.org/computational-thinking

Lorenceau, A., Marec, C. & Mostafa, T. (2019). Upgrading the ICT questionnaire items in PISA 2021 OECD, Education Working Paper No. 202.

Lix, L., Keselman, J., & Keselman, H. (1996). Consequences of Assumption Violations Revisited: A Quantitative Review of Alternatives to the One-Way Analysis of Variance "F" Test. *Review of Educational Research*, 66(4), 579-619.

Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys. 33 (1): 31–88.

Morgan, N. (2014). JavaScript for Kids. Publisher: No Starch Press.

Rigby, P., German, D., & Storey, M. (2008). Open source software peer review practices. In *Proceedings of the 30th International Conference on Software Engineering*, pp. 541-550. DOI: 10.1145/1368088.1368162

Salkind, N. J. (Ed.) (2010). Encyclopedia of research design Thousand Oaks, CA: SAGE Publications, Inc. DOI: 10.4135/9781412961288

Wing, J.M. (2006). Computational thinking. Communications of the ACM, 49(3), 33–35.