Volume 3, No: 1
April 2019
ISSN 2513-8359

International Journal of Computer Science Education In Schools

www.ijcses.org

International Journal of Computer Science Education in Schools

April 2019, Vol 3, No.1

DOI: https://doi.org/10.21585/ijcses.v3i1

Table of Contents

International Journal of Computer Science Education in Schools, April 2019, Vol. 3, No. 1
ISSN 2513-8359

Exploring problem decomposition and program development through block-based programs

Kyungbin Kwon[1]
Jongpil Cheon[2]

[1] Indiana University
[2] Texas Tech University

DOI: 10.21585/ijcses.v3i1.54

Abstract

Although teachers need to assess computational thinking (CT) for computer science education in K-12, it is not easy for them to evaluate students' programs based on the perspective. The purpose of this study was to investigate students' CT skills reflected in their Scratch programs. The context of the study was a middle school coding club where seven students voluntarily participated in a five-week coding activity. A total of eleven Scratch programs were analyzed in two aspects: problem decomposition and program development. Results revealed that students demonstrated proper decompositions of problems, which supported program development processes. However, in some cases, students failed to decompose necessary parts as their projects got sophisticated, which resulted in the failure or errors of programs. Regarding program development, algorythmic thinking had been identified as the area to be improved. Debugging and evaluation of programs were the necessary process students needed to practice. Implications for teaching CT skills were discussed.

Keywords: computational thinking, Scratch, decomposition, computer science education, block-based programming

1. Introduction

Since Wing (2006) suggested that computational thinking (CT) is "a fundamental skill for everyone, not just for computer scientists (p. 33)," many stakeholders have tried to develop a sustainable curriculum that encourages more students to learn programming earlier. However, the deficiency of K-12 computer science (CS) education is not getting better (Google & Gallup, 2015). To compensate for the lack of CS education, many researchers and teachers have offered after-school activities, such as coding clubs (e.g., Smith, Sutcliffe, & Sandvik, 2014). Researchers have suggested that young students can engage in CT concepts and practices through block-based programming (BBP), such as Scratch and Alice (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017; Sáez-López, Román-González, & Vázquez-Cano, 2016). BBP provides a visual representation of programming, which reduces the cognitive load by excluding the chances of syntax errors, using commands similar to spoken languages, providing immediate feedback, and visualizing abstract concepts (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). Because of its educational features, the use of BBP has increased in introductory CS education courses (Aivaloglou & Hermans, 2016). However, considering the limited amount of time, teachers' expertise, voluntary engagement in activities, and different skill levels among the students, there are concerns about their effectiveness (Buss & Gamboa, 2017).

CT requires problem-solving skills that involve analytical thinking to design systems (Wing, 2006). Thus, the core CT concepts, including decomposition (break problems down into smaller parts) and abstraction (model the core aspects of problems), are the target capacities of K-12 CS education (Liu, Cheng, & Huang, 2011). Although utilizing BBP has been encouraged for its effect of enhanced understanding of programming concepts, logic, and

computational practices (Sáez-López et al., 2016), there is a scarcity of studies that suggest pedagogical guidance based on students' CT skills in K-12 CS education contexts.

One of the reasons for the lack of pedagogical guidance may be due to the difficulty of evaluating CT skills that are embedded in the programs that students create. For example, a student may not be successful in decomposing the main task and developed an ineffective program that included errors. Without direct communication regarding the student's solution plan and conceptual understanding of the code, it will be difficult to pinpoint the reasons for the errors by only examining the outcome of the thinking process: successful or unsuccessful programs (Brennan & Resnick, 2012). It is also possible that multiple factors affect the problem-solving processes and the quality of the program in turn.

To evaluate CT skills, we need a precise definition and evaluation frame. Although many scholars have defined CT and identified its components (D. Barr, Harrison, & Conery, 2011; Shute, Sun, & Asbell-Clarke, 2017; Wing, 2006), it has not been sufficiently suggested how instructors can evaluate the CT concepts based on students' programs. Additionally, valid evaluation rubrics to measure computational thinking have not been established yet. As Buss and Gamboa (2017, p. 201) suggested, computational thinking is "a rich mixture of cognitive skills and attitudes" that should be evaluated from multiple aspects rather than one simple result: success or failure. To provide meaningful feedback and guidance, teachers need to assess computational thinking in detail and figure out students' misconceptions.

Considering the limited evaluations in CS education, the current study aims to examine CT skills reflected in students' programs, which will suggest an evaluation framework of CT. This study also suggests instructional strategies to be considered in secondary CS education.

2. Literature review

2.1 Computational thinking

The concept of CT has been refined through the collaboration of scholars since Wing (2006) coined the term by identifying its core elements as "solving problems, designing systems, and understanding human behavior by drawing on the concepts fundamental to computer science" (p. 33). As Wing's definition emphasizes, CT does not simply refer to computer programming skills, but is more closely related to the way we solve problems by utilizing the power of computing. From this perspective, the International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA) defined CT as a problem-solving process that includes: formulating problems, logically organizing and analyzing data, representing data through abstractions, evaluating possible solutions, automating solutions through algorithmic thinking, and generalizing solutions (D. Barr et al., 2011; CSTA & ISTE, 2011). The definition provides a framework for K-12 educators to design CT activities and evaluate CT skills. Based on this common understanding, many researchers have reached an agreement that CT involves the thought processes of decomposition, abstraction, generalization, algorithmic thinking, and debugging (Angeli et al., 2016).

Considering the context of problem-solving, we can imagine how CT is associated with students' thought processes. When students have a problem to solve or a task to achieve, they will break the problem (or the task) into smaller parts so they can manage their cognitive resources effectively (decomposition of problems). After figuring out the necessary functions or solutions of each part, they will devise a sequence of the solutions to identify the order of the actions and the conditions of control (algorithmic thinking). If students consider the efficiency and utility of the problem-solving process, they will try to create a model by extracting the fundamental characteristics of the solutions (abstraction) and generalize the solutions by parameterizing the variables (generalization). After developing the solutions, students will test whether each action corresponds to the intended instruction and fix the errors once they occur (debugging).

Although the definition of CT and its components have been refined for over a decade, there is a lack of evaluation criteria and analysis methods to reveal the levels of CT. Because CT involves problem-solving, it is difficult to measure CT through a simple test. In the following section, we will review a few trials to evaluate decomposition and program development process in problem-solving contexts.

2.2 Decomposition of problems

The core function of decomposition is to identify subtasks and define the objects and methods required in each decomposed task to solve a problem (V. Barr & Stephenson, 2011). How can we measure the decomposition process while students solve a problem? Decomposing a problem is required to design a solution. Instructors

can evaluate decomposition by examining students' solution plans. Kwon (2017) analyzed students' solution plans and identified their misconceptions of programming. The study revealed cases where students (novice programmers) did not consider all the possible solutions while decomposing a problem, failed to identify the specific functions required, and designed inefficient solutions. Students' insufficient decomposition of problems could be attributed to the lack of knowledge schemas and the higher cognitive load required in their thought processes (Kwon, 2017; Robins, Rountree, & Rountree, 2003).

2.3 Program development (abstraction, generalization and algorithmic thinking)

Even when students have a clear plan, they often demonstrate an iterative cycle of developing codes: trying out codes, changing plans, integrating new ideas, and so on (Brennan & Resnick, 2012). Thus, if we evaluate CT only based on the knowledge-based concepts (e.g., sequences, loops, conditional, and events), we will not be able to evaluate how students use or apply the knowledge into their programs (Davies, 1993).

Students can demonstrate their CT skills during the process of developing programs and through artifacts (Lee, 2010). In this sense, scholars have recently suggested various ways to evaluate CT based on student-developed programs. For example, Moreno-León, Robles, and Román-González (2015) introduced Dr. Scratch (http://www.drscractch.org) that automatically evaluates Scratch programs. Dr. Scratch allows teachers and students to evaluate programs in terms of the CT concepts: abstraction and problem decomposition, logical thinking, synchronization, parallelism, algorithmic notions of flow control, user interactivity, and data representation. It is noteworthy that Dr. Scratch assesses CT concepts by evaluating students' programs rather than asking their knowledge (e.g., Grover & Basu, 2017; Meerbaum-Salant, Armoni, & Ben-Ari, 2013). However, there is a limitation in that Dr. Scratch does not consider the purpose and functionality of the codes that are related to the effectiveness and efficiency of programs.

Recently, Kwon, Lee, and Chung (2018) evaluated CT by manually analyzing students' Scratch programs in consideration of programming goals and tasks. They found that students often added unnecessary sets of programs, which caused a redundancy of codes that increased the complexity and the chances of errors. They also suggested the positive relation between the ability to decompose problems and the quality of the solutions. It is suggested that evaluating CT skills in authentic tasks where students apply the CT skills to solve problems.

3. Purpose of study

The current study aimed to examine students' Scratch programs from two perspectives: problem decomposition and program development (abstraction, generalization, and algorithmic thinking). The findings of the study would provide insight to build an evaluation framework for CT. This study, therefore, addressed the following research questions:

1. How do students decompose tasks for Scratch programs?
2. How do students create Scratch programs by utilizing abstraction, generalization, and algorithmic thinking?

4. Method

4.1 Participants

Seven middle school students (six girls and one boy) participated in the after-school coding event. All students learned in "Hour of Code" during their school curriculum. Four students had experienced coding with BBP, such as Scratch or Tynker, before participating in the event. Students rated their coding skill as basic (2.1 out of 5 in average) at the beginning of the event. They were not given compensation for their participation in the study. The study was approved by the University Institutional Review Board (#1802262819) and public school corporation.

4.2 Context of Learning

Partnering with the middle school coding club, a researcher (the first author) from a university in the Midwest developed the curriculum for the five-week coding event: "Going Beyond the Hour of Code." The event was designed for middle school students to learn CT skills by developing games, quizzes, and applications using a BBP called Scratch (https://scratch.mit.edu). The curriculum was designed to let students represent a problem and solve it using a computer program, break a problem down into smaller parts, design a series of instructions to formulate the solution, and apply problem-solving skills to a wide variety of problems. No prior coding experience was required for the students during the recruiting process. Before the event, the students participated in an

introductory session that explained the purpose of the event and an opportunity to participate in the research. Table 1 describes the contents of the curriculum.

Once the students gathered in the computer lab, the researcher explained the theme for the week, and demonstrated how to create a corresponding Scratch project. The researcher emphasized the main CT skills during the demonstration and encouraged students to develop a Scratch project that fulfilled the requirements. Typically, the researcher led the demonstration for 20 to 30 minutes, and students had 45 to 55 minutes to create their own Scratch project.

Table 1. Overview of Coding Event Curriculum

Week	Theme	CT skills	Required components	Tasks to achieve
1	Dance (Loop)	Develop a program repeating particular actions by utilizing loops	• Changing the costumes of the sprite • Playing music • Changing the background • Moving the sprite repeatedly	Create a dancing spite and play music. Change the background appropriately.
2	Maze (Conditions)	Develop decision making skills by utilizing the *if* block	• *Motion* blocks • *Sensing* block (touching color, touching "object") • *If* block (utilizing sensing and motion blocks)	Create a maze game. Let the sprite (mouse) come back to the beginning point when either it hits a maze or another sprite (cat).
3	Catch & Avoid (Data)	Define variables and use them for decision-making processes	• Making a variable • Updating values of the variable • Examples of using variables in the *if* block	Create a game that increases the scores when the user completes a task. Specify how many trials (lives) that the user has.
4	Quiz (Patterns)	Receive user's input Use a *broadcast* block to control other sprites	• Ask for user responses and receive inputs • Make a decision based on the inputs • Broadcast commands to other sprites	Create a quiz game Decide whether the answer is correct and increase the score accordingly. Change other sprites' costumes based on the answer.

4.3 Data

Each week, students submitted their Scratch projects in a learning management system, Canvas. A total of 18 projects were collected, but the researchers only analyzed the projects of the students who submitted an informed consent form. So, a total of 11 projects from four students were analyzed for this study. The names are pseudonyms.

4.4 Analysis

To have an in-depth understanding of the Scratch programs that the students created, we analyzed them in terms of decomposition and program development reflected in the programs. The unit of analysis was a semantic unit that included one or several code blocks executing a particular task. To identify decomposition, we considered the alignments of sub-tasks and sets of blocks. To evaluate program development, we evaluated sets of blocks in terms of their functionality and efficiency.

5. Results

Scratch projects were analyzed based on two CT aspects: decomposition and developing programs (abstraction, generalization, and algorithmic thinking). The results are presented by the weekly theme.

5.1 Week 1 Loops: Decomposition

5.1.1 Changing a sprite's look or position

All the projects showed that the students successfully identified the required tasks to decompose (see Table 2).

Table 2. Week 1 decomposition and program development

Dance (Loop)	Make sprites dance by using repeat blocks
Decomposition of tasks	• Changing a sprite's look or position • Making a meaningful story
Program development	• Creating a set of blocks (Motion or Look) to repeat • Using a repeat block

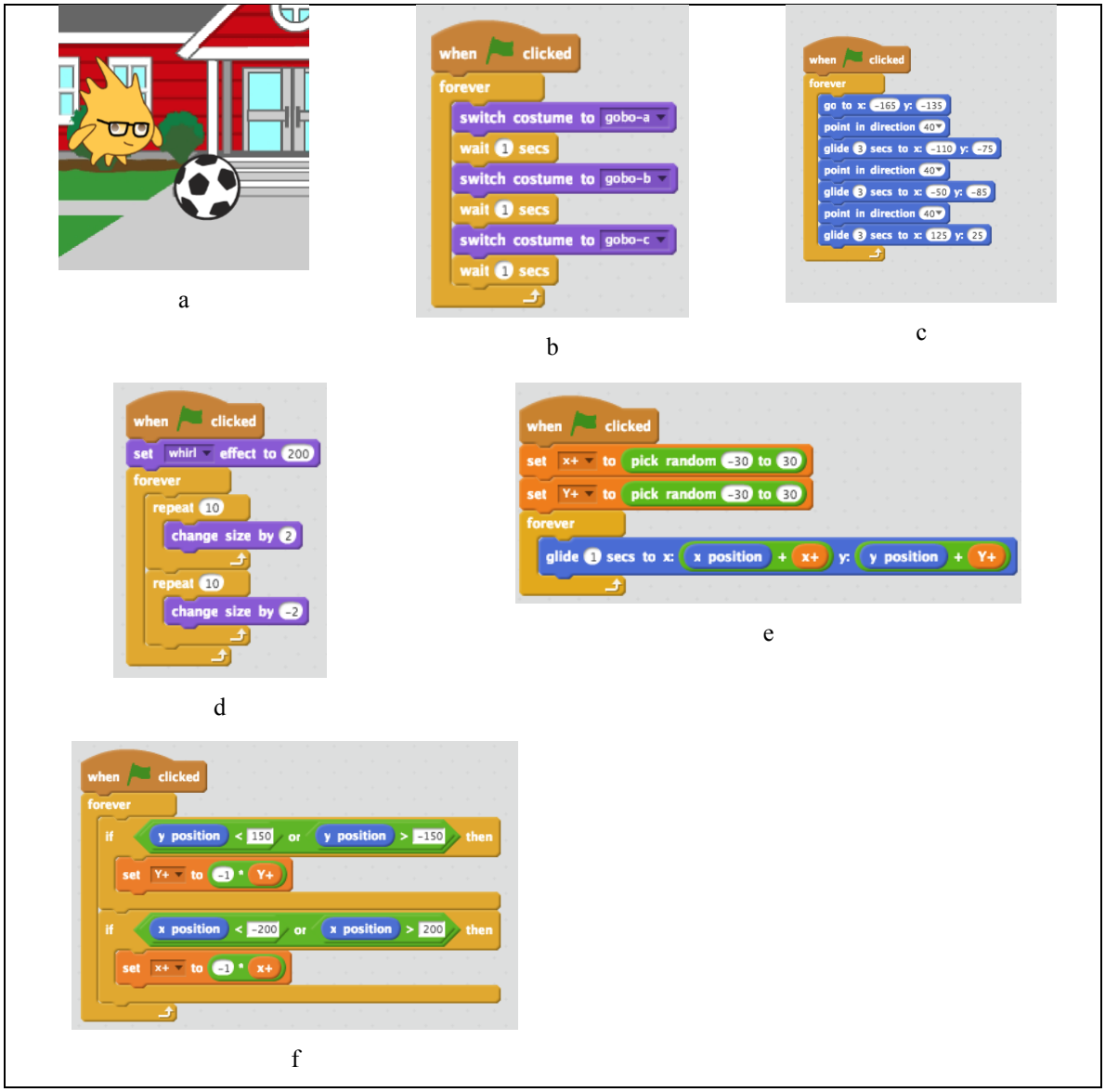

Figure 1. Code blocks of week 1

5.2 Week 1 Loops: Program development

5.2.1 Successful aspects

Boa, and Emily used the 'switch costume' block to make the sprites dance, while Susan changes a ball's size and location using the 'change size' and 'position' blocks. Most of the students successfully utilized the 'forever' block

to repeat a set of codes to move a sprite.

We found that a student discovered an alternative way to move a sprite. Kathy used the 'switch costume' block to make a moving sprite dribble a ball (see Figure 1-a). She changed the location of the sprite in the canvas and switched its costumes, which seemed to make it move (see Figure 1-b). After she synchronized the locations of two sprites (main character and its glasses), she tried to adjust the ball's location. At that moment, she did not know "move" block but, soon after, realized the block and used it for next tasks (see Figure 1-c).

5.2.2 Issues to be considered

We found two issues in Susan's project. First, she used different blocks to move the ball and change its size (see Figure 1-d). She already knew how to use condition blocks to check the position of a sprite to control it from crossing the boundaries of the stage (see Figure 1-f). She also used variables and randomized position values (see Figure 1-e). The code blocks demonstrated her abstraction skills to program the decomposed tasks. However, there was an error in the 'if' statement: x position < -200 or > 200 (see Figure 1-f). In order to limit the range of the sprite, the statement should be x position > -200 or < 200. It seemed that Susan did not check the logical expression and failed to recognize the error.

5.3 Week 2 Conditions: Decomposition

To make a maze game, they needed to identify the events that would occur during the game, including allowing the user to move the main sprite with keystrokes, resetting the game when the sprite touched a maze or was hit by an object, and finishing the game when a sprite reached the finish line. The Scratch projects showed that Kathy and Susan decomposed the necessary events accordingly (see Table 3). Additionally, Kathy developed two different stages in her maze game that was an advanced feature among other projects.

Table 3. Week 2 decomposition and program development

Maze (Conditions)	Develop a maze game that makes decisions as pre-defined events occur by utilizing conditions
Decomposition of tasks	Identifying required events with conditions • Moving sprites according to keystrokes • Resetting the game when being hit by objectives • Resetting the game when touching the maze • Ending a game when arrive at the finish line
Program development	Using forever and if blocks to create the event handlers • with arrow keys • with touch

5.4 Week 2 Conditions: Program development

Regarding the decomposed events, students should utilize the 'forever' and 'if' blocks with two different conditions, such as the 'when a key pressed' and 'touching' blocks, to develop a maze game.

5.4.1 Successful aspects

Kathy utilized a 'forever' block to nest several 'if' blocks that identified the events, such as the "key pressed", "touching a color", and "touching another sprite" blocks (see Figure 2-a). All the event handlers in her program shared the same structure (if blocks nested in forever block that monitored a particular event). Thus, Kathy was able to abstract the structure of the codes for each event. Susan used a 'broadcast' block to reset the game when the main sprite touched the maze (blue color) (see Figure 2-c and 2-f).

5.4.2 Issues to be considered

Susan did not use a 'forever' block to check if the main sprite touched a color. Instead, she used a pre-defined event block 'When key pressed' with the 'if' block (see Figure 2-c). The way Susan utilized the event handlers required four duplicated codes for four different key presses. She also had an unnecessary block, 'wait 0.6 secs.' The results suggested that Susan failed to find an efficient way to check for the specific condition (i.e., touching

color).

Susan wanted the sprite to say, "You Win!" when it touched the ending spot that was a green dot. However, she used the 'repeat until <touching color green>' block (see Figure 2-e). It would be possible that the 'say' block was inside of the 'repeat until' block, which resulted in showing "You Win!" from the beginning repeatedly. Because she did not resolve the issue, she moved the say block to out of the 'repeat until' block. The issue was related to the lack of understanding of envent handlers. She should have used a 'forever' and 'if' block to make the event handler work as intended.

5.4.3 Additional Findings

Because Kathy developed multiple stages, she needed to change the backdrop and sprites accordingly. She wanted to hide an object during the final stage came after the main sprite passed the maze. However, she could not hide the objects used in the maze. It was related to the synchronization of codes in Scratch. The researcher asked for her intention and suggested that she use the 'broadcast' block, which she hadn't learned at that moment. She easily understood the use of the 'broadcast' block to make the objects disappear after a short conversation with the researcher (see Figure 2-b). This finding suggested the importance of tailored guidance according to student needs for discovery learning. As mentioned, Kathy, including other students, tried to discover solutions that they had not learned yet while developing programs.

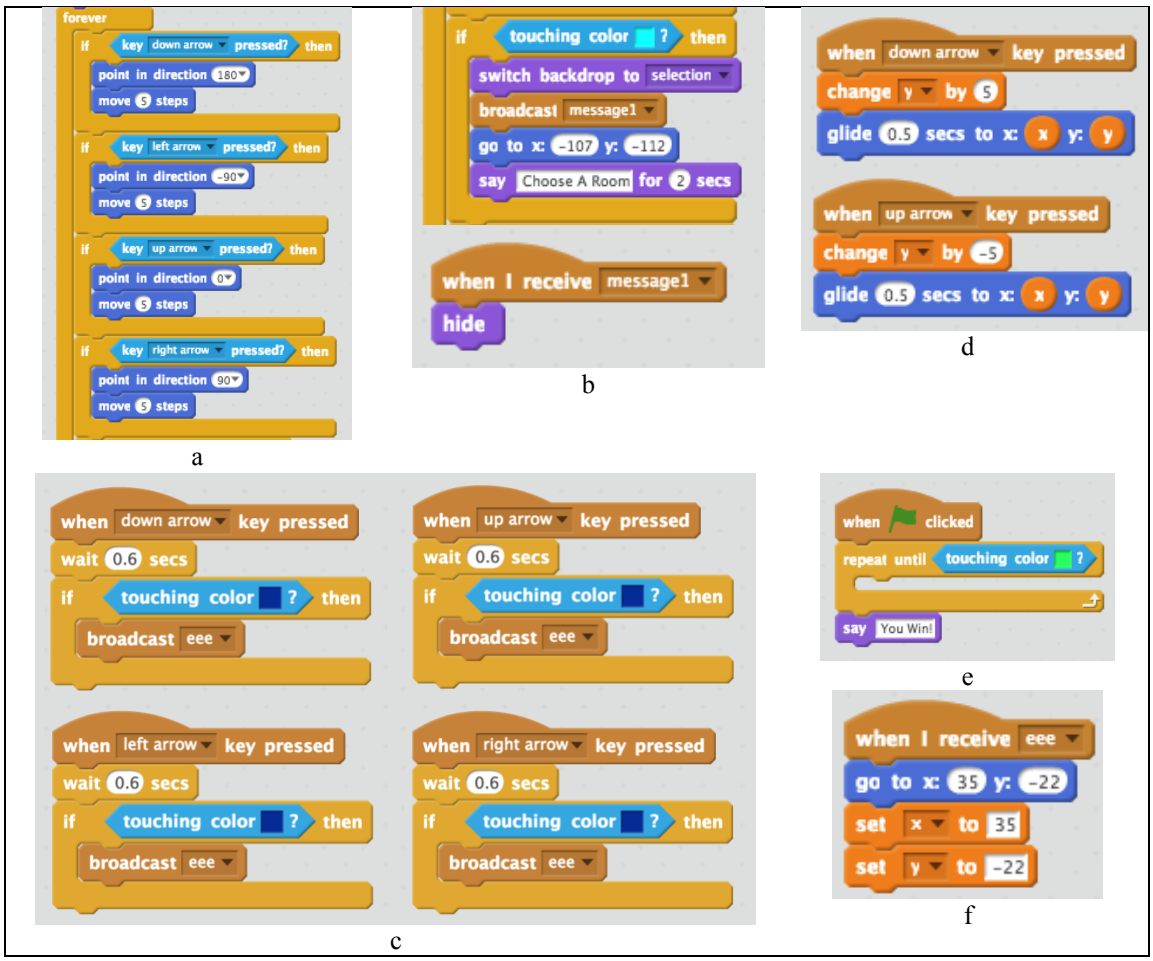

Figure 2. Code blocks of week 2

5.5 Week 3 Data: Decomposition

The primary objective of week 3 was to update user scores according to the user's performance in a game. The decomposed tasks to create this game were defining a mission to accomplish (e.g., catching an objective while avoiding another object), gaining or losing points and finishing a game according to the score (such as changing the levels of game or success/failure based on the accumulated score), showing or hiding an object in random

locations, and moving sprites with keystrokes (see Table 4). Both Kathy and Susan were able to decompose the main task to smaller sub-tasks, which allowed them to organize code blocks based on the sub-tasks. Susan, however, did not identify the condition to finish the game.

Table 4. Week 3 decomposition and program development

Catch & Avoid (Data)	Develop a game that saves scores and makes a decision based on the scores.
Decomposition of tasks	• Defining a mission (e.g., touch or avoid specified objects) • Gaining or losing a score when accomplish or fail a mission • Moving sprites according to keystrokes • Showing/hiding objects in random locations • Finishing game according to the score

Program development	• Using variables to save and update values
	• Using forever and if blocks to create the event handlers
	  ○ with arrow keys
	  ○ with touch
	• Using random block to display objects in random places
	• Using if block to check the condition to finish a game

5.6 Week 3 Data: Program development

5.6.1 Successful aspects

The analysis of code revealed that Kathy considered the efficiency of the codes and selected a proper way to fulfill the objective. In contrast of the previous program, Kathy separated her code into five sets according to their purposes, such as touching obstacles, accomplishing a level, changing the stages of game, and ending the game (See figure 3-a). Although the researcher did not explicitly mention the concept of parallelism while demonstrating an exemplary program, she grasped the concept and organized her codes as the tasks decomposed.

Kathy realized that there were several different ways to achieve a particular task. For example, she used the value of variables to determine when a certain sprite should disappear. In her code, all the sprites were set to disappear when the value of "life" became less than 1 (see Figure 3-b). She made multiple stages of the game, which required changing the backdrops and sprites accordingly. She utilized 'broadcast' blocks to fulfill the requirement. The 'broadcast' blocks sent out messages when a level was completed (e.g., Level 1: Completed) and each sprite reacted on the messages (see Figure 3-c and 3-d).

Susan defined a variable and used a 'random' block to assign random values to the variable. By using this method, she could change the backdrops randomly (see Figure 3-e). In this process, Susan gave each backdrop a numerical value that cooperated with a random number. It is noteworthy that she utilized the name of the backdrops for a computational purpose.

5.6.2 Issues to be considered

Kathy did not figure out the conditional logics while developing multiple decision-making processes by utilizing 'if' blocks. As figure 3-c illustrates, she included three 'if' blocks to identify the criteria of three decision making points: Score > 12, Score > 24, and Score >30. Considering the flow of game, we assume that Kathy wanted to identify the threshold of the levels as follows: if Score = 13, Score = 25, and Score = 31. Kathy might have assumed that the computer would run the second condition when the score got to 25. However, the 'if' blocks in the first conditional statement would never proceed to the next conditions because the first condition would be "true" for other conditions. The results suggests that students can make mistakes when they assume that a computer works as humans think (Kwon, 2017; Pea, 1986). To overcome egocentrism in programming (Pea, 1986), it is necessary for students to distinguish the intent of the programmer and the actual instructions that are explicitly programmed into the code.

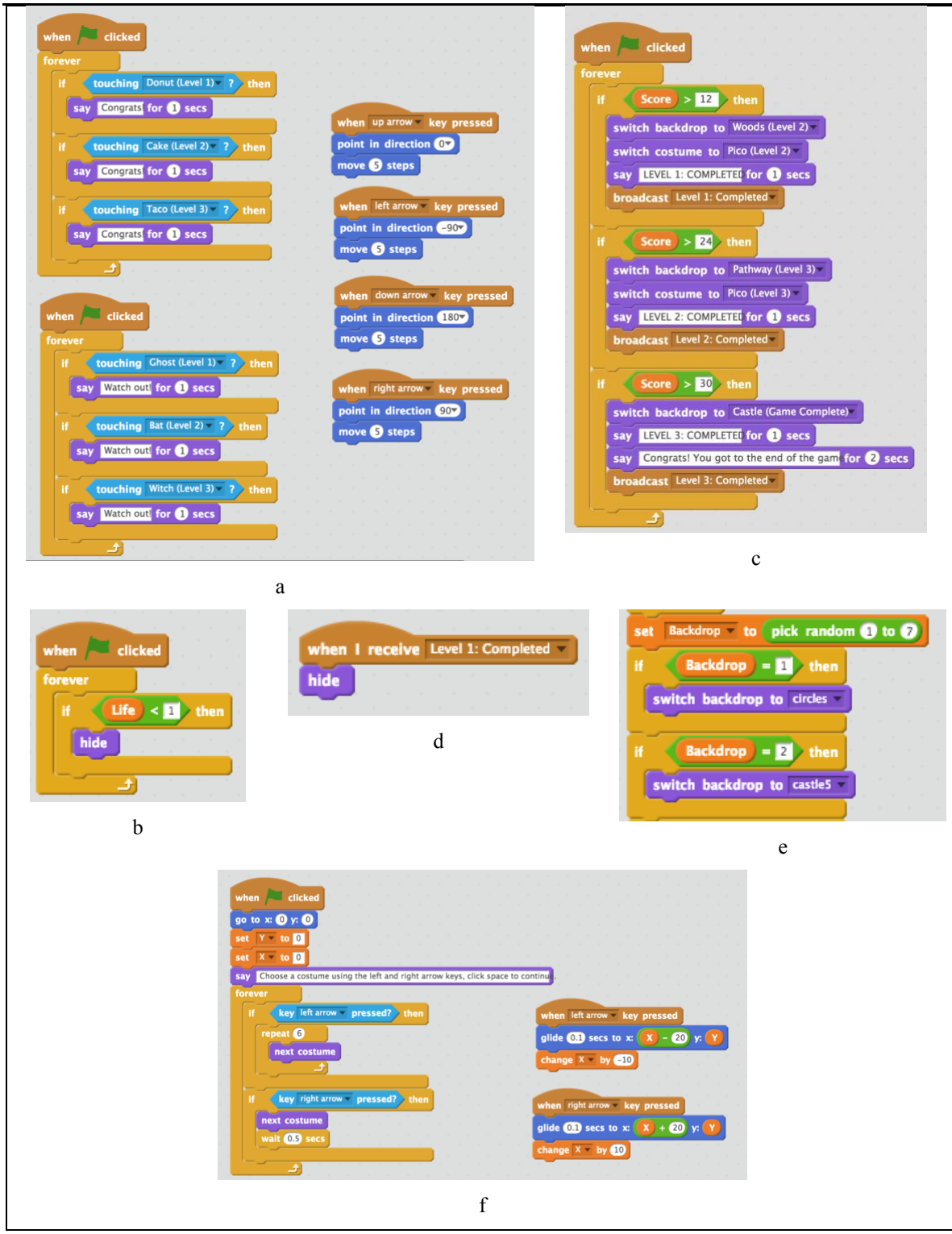

Figure 3. Code blocks of week 3

Susan committed an error that caused a conflict for the arrow keys. She used the arrow keys to control the movement of the sprite, while assigning the left and right keys to change costume of the sprite as well (see Figure 3-f). Although she successfully decomposed the tasks, she failed to consider the whole program and did not figure out how the code would conflict when a single key had two functions. Considering the limitation of novice programmer's cognitive capacity, providing a concrete model representing the codes and encouraging students to describe how the codes will run in their words will be beneficial (Mayer, 1981).

5.7 Week 4 Patterns: Decomposition

To create a quiz program, students needed to consider the following tasks: asking questions, receiving user inputs, evaluating the inputs, and providing feedback (see Table 5). Because there were multiple questions in a quiz set, the tasks should be repeated with the same process. To create an efficient program, students should identify the patterns of the tasks and develop code blocks that could be used multiple times with different contents, such as questions and correct answers. As an exemplary case, Kathy created four different categories with different sets of questions. Kathy successfully decomposed the tasks and identified patterns to develop an efficient program. However, Emily failed to organize the sub-tasks required to check user-entered answers and provide feedback.

Table 5. Week 4 decomposition and program development

Quiz (Patterns)	Make a quiz that asks multiple questions and provides feedback accordingly
Decomposition of tasks	• Asking questions • Checking answers entered by users • Providing feedback according to the answers
Program development	• Using ask block to show questions • Using if-else block to compare user inputs and correct answers • Using broadcast block or other blocks to provide feedback

5.8 Week 4 Patterns: Program development

5.8.1 Successful aspects

Kathy successfully demonstrated her abstraction skills in her coding. As Figure 4-a illustrates, she used three sets of blocks for one question: 'ask', 'if-else', and 'broadcast' blocks. For example, the 'if-else' block evaluated user input and decided whether it was correct or incorrect. The set of these three blocks were repeated for the other questions. Her use of 'broadcast' blocks demonstrated her abstraction skill. As possible results of user input, Kathy defined two 'broadcast' blocks: "Correct Answer" and "Wrong Answer" (see Figure 4-b). As the names of the blocks were implied, one sent a message that the user input was correct, while the other sent a message that the user input was incorrect. It is noteworthy that Kathy could identify these patterns and developed reusable code blocks for every question. Her conditional expression in the 'if-else' block demonstrated her understanding of conditional logic. Distinctly, she used the 'logical OR' expression to consider multiple correct answers (i.e., see Figure 4-a), which suggested her advanced computational thinking skill in developing "short" programs while considering "multiple cases."

5.8.2 Issues to be considered

In contrast, Emily failed to abstract the primary structure of code blocks and made the program complex and inefficient. As Figure 4-c illustrates, she did not identify the patterns of the tasks as Kathy did. Without clearly decomposing the tasks, she used the nested 'if-else' blocks to consider whether the user-entered answers were correct or not. It also seemed that she was distracted by other minor features, including the size or costume of the sprites that were not the primary tasks of the program. The analysis of her program suggests that students may develop inefficient and more complex programs when they fail to decompose the tasks and develop a program without a clear plan, such as flowchart.

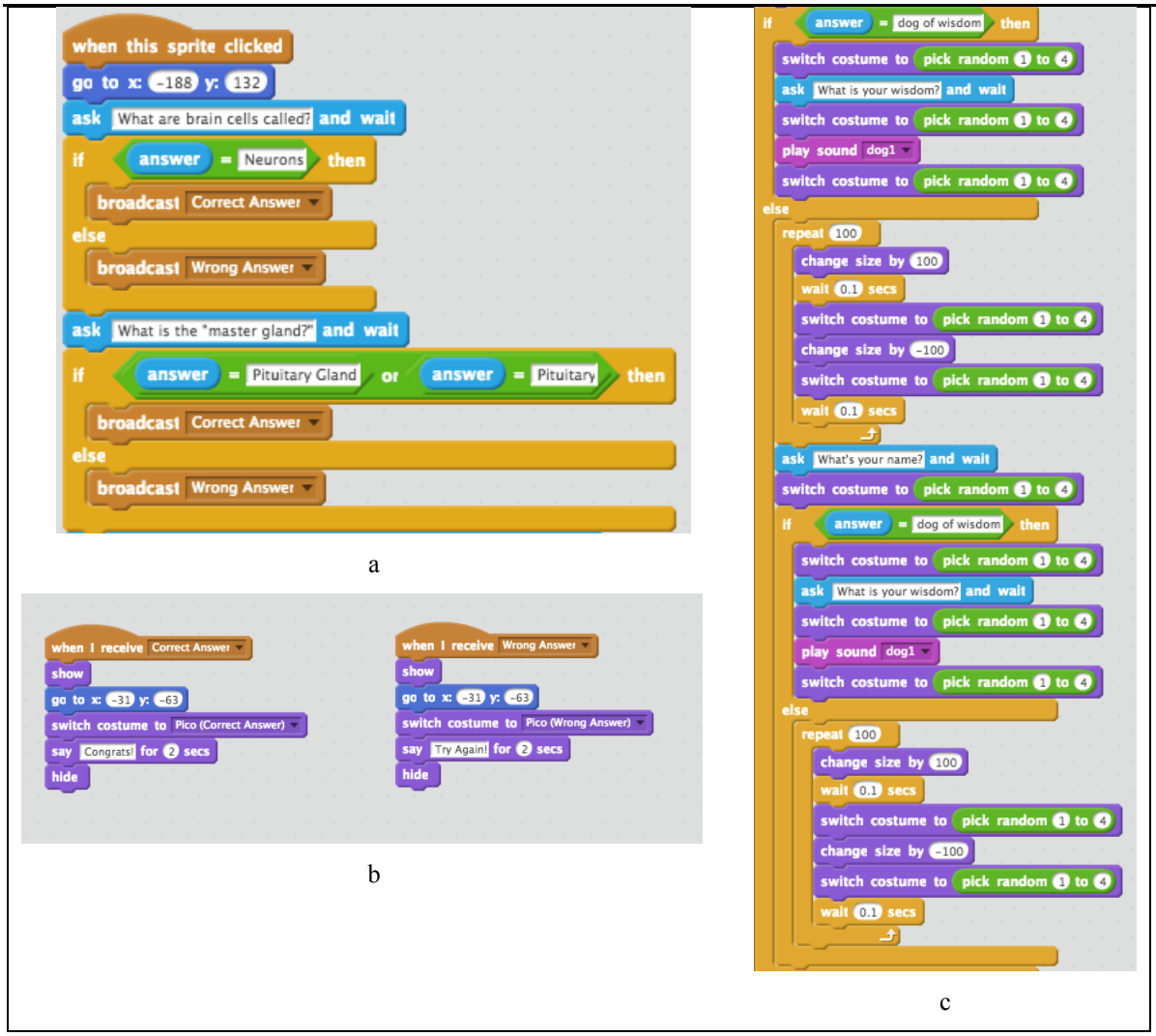

Figure 4. Code blocks of week 4

6. Discussion and conclusion

The findings showed that students quickly grasped the concepts of sequence including repeat and decomposed the required subtasks for simple projects. The abstraction and parallelism skills have been progressively improved as they practiced. On the other hand, some students failed to decompose sub-tasks for sophisticated games and debug errors in their codes. If they tested their program more often, they would have a chance to fix the errors. It seems challenging for them to make a conditional statement more efficiently (e.g., expressing multiple conditions exclusively). In addition, the condition statements with operators were not logical to determine the correct conditions. The challenges found in Scratch programs yield numbers of implications for teaching CT skills through programming based on the results. The implications include instructional considerations for 1) planning activities, 2) decomposition, 3) logical thinking, and 4) debugging.

First, guided planning activities are necessary at the beginning of programming. As Emily's project showed, students can focus on a specific task without considering the purpose of the program. We often observed that students started programming without a clear plan and tinkered with code blocks by trial and error. We suggest that students need to learn how to sketch out their story, and that it should be the first step to design a program after learning the core concepts of programming (Kim, Kim, & Kim, 2013). Instructional strategies for planning, such as creating a story synopsis or storyboard by writing or drawing, can be adopted (Brennan & Resnick, 2012).

Second, students need to be trained to decompose a task properly. The findings showed that students struggled with decomposing tasks as their project got sophisticated. For example, Susan did not set the end of the game, and Emily failed to identify the sub-tasks of the quiz game. Decomposition is an essential process to represent problems and identify the tasks to achieve by considering the events, decision-making points, and functions of the codes when designing a program (Kazimoglu, Kiernan, Bacon, & MacKinnon, 2012). Without proper decomposition, students cannot design appropriate sequences of codes, consider parallelization of multiple codes,

and develop modularized code blocks. The decomposition process should be emphasized when teaching programming so that students can design small sets of code according to the sub-tasks. Possible activities to help decomposition is to create a decomposition chart, or flowchart, using graphic representation (Robins et al., 2003). Students can use a worksheet or template to practice the decomposition process.

Third, students need scaffolding that supports their logical thinking in developing programs. The results revealed some issues were related to reasoning in program development. In week 2, for example, Susan duplicated codes when checking if the sprite touched a specific color and failed to meet the goal of using 'forever' and 'if' blocks. In week 3, Kathy was not able to figure out the right conditional logic to determine moving to next level due to the error in 'if' conditions. Additionally, Susan had the conflict with the multiple functions of the keystrokes to move the sprit and change its costume. In week 4, Emily could not program correctly nested 'if' block to check text users entered. Lack of knowledge about the relevant blocks could cause the mistakes; however, the lack of logical thinking could yield the errors because most of them failed to write correct logical expressions and develop complex conditional structures even though they used the proper blocks. Therefore, we should guide students to work on core logic by practicing a simpler version first. As the elaboration theory advocates (Reigeluth, 1999), students need to practice simple tasks involving a specific reasoning skill earlier. Also, a condition chart or pseudo code could aid with logical reasoning to structure decomposed tasks and determine relevant conditions (Kim et al., 2013).

Lastly, instructors need to emphasize debugging practice to students. By only analyzing Scratch programs, we were not able to examine students' debugging process. However, we found that students often overlooked or ignored errors that could be detected by simple tests. For example, the errors of conditional statements and the conflict of the same keystrokes could have been caught if the students tested the programs. Debugging includes not only fixing errors but also increasing the efficiency of code (Robins et al., 2003). The findings suggest that students need to evaluate their programs efficiency as well. It could be done by sharing projects with peers and evaluating programs together (Wang, Li, Feng, Jiang, & Liu, 2012), such as pair programming.

Although there are many ways to measure computational thinking, the current study has explored the way to analyze Scratch programs based on two major computational thinking components (i.e., decomposition and program development). The results revealed the challenges students faced during the design and development phases of their programs, and instructional strategies were discussed regarding facilitating planning activities, decomposition, logical thinking and debugging. However, the measurement of this study is limited, and future research based on the limitations should be noted. First, analyzing the larger samples of Scratch programs will give us more accurate pictures of students programming patterns and mistakes. Second, other supplementary data would provide the students' thinking process in detail (Lye & Koh, 2014). It limits the understanding of the programming process (e.g., pattern recognition or debugging) by only analyzing the final products (programs). As we suggested earlier, working documents, such as story synopses, decomposition charts, condition charts or reflection journals (Robertson, 2011), would be not only helpful for developing programs to students, but also be significant data source to the instructors for in-depth analysis of computational thinking. Third, the benefit of computational thinking should be investigated. For example, the effect of computational thinking skills on problem-solving skills have not been empirically tested, and further research on the relationships among sub-computational thinking components should be considered. The findings of the further research will contribute to better instructions that will enhance computational thinking.

References

Aivaloglou, E., & Hermans, F. (2016). *How Kids Code and How We Know: An Exploratory Study on the Scratch Repository*. Paper presented at the Proceedings of the 2016 ACM Conference on International Computing Education Research, Melbourne, VIC, Australia.

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 Computational Thinking Curriculum Framework: Implications for Teacher Knowledge. *Journal of Educational Technology & Society, 19*(3), 47-57.

Barr, D., Harrison, J., & Conery, L. (2011). Computational Thinking: A Digital Age Skill for Everyone. *Learning & Leading with Technology, 38*(6), 20-23. doi:citeulike-article-id:10297515

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *ACM Inroads, 2*(1), 48-54. doi:10.1145/1929887.1929905

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: blocks and beyond. *Communications of the ACM, 60*(6), 72-80. doi:10.1145/3015455

Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking.* Paper presented at the Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada.

Buss, A., & Gamboa, R. (2017). Teacher Transformations in Developing Computational Thinking: Gaming and Robotics Use in After-School Settings. In P. J. Rich & C. B. Hodges (Eds.), *Emerging Research, Practice, and Policy on Computational Thinking* (pp. 189-203). Cham: Springer International Publishing.

CSTA, & ISTE. (2011). *Operational definition of computational thinking for Ke12 education*. Retrieved from https://c.ymcdn.com/sites/www.csteachers.org/resource/resmgr/CompThinkingFlyer.pdf

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

Google, & Gallup. (2015). *Searching for Computer Science: Access and Barriers in U.S. K-12 Education*. Retrieved from https://goo.gl/oX311J

Grover, S., & Basu, S. (2017). *Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic*. Paper presented at the Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Kazimoglu, C., Kiernan, M., Bacon, L., & MacKinnon, L. (2012). Learning Programming at the Computational Thinking Level via Digital Game-Play. *Procedia Computer Science, 9*, 522-531. doi:10.1016/j.procs.2012.04.056

Kim, B., Kim, T., & Kim, J. (2013). Paper-and-Pencil Programming Strategy toward Computational Thinking for Non-Majors: Design Your Solution. *Journal of Educational Computing Research, 49*(4), 437-459. doi:10.2190/EC.49.4.b

Kwon, K. (2017). Novice programmer's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools, 1*(4), 14-24. doi:10.21585/ijcses.v1i4.19

Kwon, K., Lee, S. J., & Chung, J. (2018). Computational concepts reflected on Scratch programs. *International Journal of Computer Science Education in Schools, 2*(3). doi:10.21585/ijcses.v2i3.33

Lee, Y. (2010). Developing computer programming concepts and skills via technology-enriched language-art projects: A case study. *Journal of Educational Multimedia and Hypermedia, 19*(3), 307-326.

Liu, C.-C., Cheng, Y.-B., & Huang, C.-W. (2011). The effect of simulation games on the learning of computational problem solving. *Computers & Education, 57*(3), 1907-1918. doi:10.1016/j.compedu.2011.04.002

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior, 41*, 51-61. doi:10.1016/j.chb.2014.09.012

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education, 10*(4), 1-15. doi:10.1145/1868358.1868363

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys, 13*(1), 121-141. doi:10.1145/356835.356841

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

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*(46), 1-23.

Pea, R. D. (1986). Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research, 2*(1), 25-36. doi:10.2190/689T-1R2A-X4W4-29J2

Reigeluth, C. M. (1999). The elaboration theory: Guidance for scope and sequence decisions. In C. M. Reigeluth (Ed.), *Instructional design theories and models: A new paradigm of instructional theory* (Vol. 2, pp. 425-453). Hillsdale, NJ: Lawrence Erlbaum Associates.

Robertson, J. (2011). The educational affordances of blogs for self-directed learning. *Computers & Education, 57*(2), 1628-1644. doi:10.1016/j.compedu.2011.03.003

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172. doi:10.1076/csed.13.2.137.14200

Sáez-López, J.-M., Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using “Scratch” in five schools. *Computers & Education, 97*, 129-141. doi:10.1016/j.compedu.2016.03.003

Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review, 22*, 142-158. doi:10.1016/j.edurev.2017.09.003

Smith, N., Sutcliffe, C., & Sandvik, L. (2014). *Code Club: bringing programming to UK primary schools through*

Scratch. Paper presented at the 45th ACM Technical Symposium on Computer Science Education (SIGCSE 14), Atlanta, GA.

Wang, Y., Li, H., Feng, Y., Jiang, Y., & Liu, Y. (2012). Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education, 59*(2), 412-422. doi:10.1016/j.compedu.2012.01.007

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35. doi:10.1145/1118178.1118215

International Journal of Computer Science Education in Schools, April 2019, Vol. 3, No. 1
ISSN 2513-8359

An Overview of Computational Thinking

Sibel Kılıçarslan Cansu, PhD [1]

Fatih Kürşat Cansu[2]

[1] Abant İzzet Baysal University Faculty of Natural Sciences

[2] Bahçeşehir University Institute of Educational Sciences

DOI: 10.21585/ijcses.v3i1.53

Abstract

Computers and smart devices have become ubiquitous staples of our lives. Computers and computer-controlled devices are used in all industries from medicine to engineering, and textile production. One field where computers have inevitably spread into is education, and one pre-requisite of controlling computers, or increasing the level and efficiency of our control over them, is making human-computer interaction as efficient as possible. This process of efficient and effective computer use, known as "Computer-like Thinking" or "Computational Thinking", is seen as a field with the potential to support individual and societal development in our rapidly progressing world and to provide significant economic benefits. The fundamental concepts and scope of this field have been delineated in diverse manners by different researchers. Similarly, researchers have also advanced distinct critical viewpoints towards and potential benefits of computational thinking. This study aims to first define the concept of computational thinking by referencing source literature, then analyze the aims of certain criticisms of the field, and discuss the fundamental elements of computational thinking and contemporary research on these elements.

Keywords: computational thinking, computer-like thinking, computational-informatic thinking

1. Introduction

"Computer" as a word references a device that "computes", localized into Turkish as "bilgisayar" by Prof. Dr. Aydın Köksal (Keser,2011: p.88). Yet it is difficult to claim the same about "computational thinking", which is localized in a number of ways by researchers. Özden et al. (2015) use "*bilgisayarca düşünme",* whereas Yesan, Özçınar and Tanyeri (2017) prefer "*hesaplamalı düşünme".* Çınar and Tüzün (2017), meanwhile, used "*bilgi sayımsal düşünme"* and "*bilgi işlemsel düşünme"* in their paper. This study will primarily use "*bilgi işlemsel* düşünme" (Computational Thinking). The presence of such diverse localization attempts is natural. As Piaget has (Bringuier, 1980: p.57) specified, definition of terms comes after the creation of terms in scientific research. The novelty of this field, leading to a lack of uniformity in jargon and everyday divergence of terms in common usage, may be the explanation of this phenomenon. A similar differentiation is observed in the computer science / informatics divide separating researchers in the field. Whereas European sources prefer the term "informatics", putting information before the devices used to process it; American researchers seem to prefer "computer science" as their term for this field (Kalelioğlu, Gülbahar and Kukul, 2016). Nonetheless, despite differences in terminology, it is observed that the fundamental focus of this field is the basic principles of computer science and their interaction with mankind.

2. The History of Computational Thinking

While computational thinking is widely considered to have begun by Wing's (2006) article on the subject, it was first referenced by Papert (1996), as "procedural thinking". Papert, then in MIT's Department of Mathematics, in the course of his research on computer and software usage in solving geometric problems claimed that computational thinking could be employed in defining the relationship between a problem and its solution and the structuring of data. Papert and his colleagues had developed the LOGO programming language in the 1960's. The main aim of this language was aiding students in thinking mathematically and logically. LOGO was at its core a constructivist language, accepting learning to be a fundamentally individual activity and explaining it in Piagetian terms. Papert (1991: p.1)'s individualization of this concept resulted in the notion of learning-by-making. Papert's adoption of this philosophy is not surprising, considering his experience working alongside Piaget in the Centre of Genetic Epistemology in Geneva between 1958 and 1963. LOGO was thus designed as an environment conducive to and supportive of Piagetian learning (Logo, 2015).

Figure 1. Seymour Papert and LOGO-based robot Turtle.

LOGO and the constructivist ethos behind it were considered to have the potential to transform education when the language was first introduced. This potential did not come to life however, as constructivism gradually lost traction in the education systems of the UK and the USA (Agalianos, Noss, and Whitty, 2001: p.497). This loss was not unprecedented, as other programming languages such as PLATO (Programmed Logic for Automatic Operations), CAI (Computer Assisted Instruction), CBT (Computer Based Training) and CAL (Computer Assisted Learning) also faced the same fate (Etherington, 2017).

3. Defining Computational Thinking

As computational thinking is a newborn field, its definition varies from researcher to researcher. Due to this variation between academics, this paper will consider practical definitions offered by organizations such as ISTE (International Society for Technology in Education) and CSTA (Computer Science Teacher Association) in addition to those determined by the academics themselves. Wing (2006, p.33) defines computational thinking as "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science.". However, after further revisions [as the original article was 4 pages long and many topics were not fully explored.] a different definition was accepted in 2011. According to Wing (2011), computational thinking is defined as "Computational thinking is the thought processes involved

in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent.". Table 1 showcases the various definitions of computational thinking employed by the contemporary academia.

Table 1. Contrasting Definitions of Computational Thinking.

Definition	Source
...the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent.	(Cuny, Snyder, Wing, 2010 akt. Wing, 2011, p.20)
Computational thinking is the thought processes used to formulate a problem and express its solution or solutions in terms a computer can apply effectively.	Wing (2014)
The mental process for abstraction of problems and the creation of automatable solutions.	Yadav et al. (2014)
Computational thinking is the process of recognising aspects of computation in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes.	Furber (2012)
Computational thinking has a long history within computer science. Known in the 1950s and 1960s as "algorithmic thinking," it means a mental orientation to formulating problems as conversions of some input to an output and looking for algorithms to perform the conversions. Today the term has been expanded to include thinking with many levels of abstractions, use of mathematics to develop algorithms, and examining how well a solution scales across different sizes of problems.	Denning (2009)
...[Computational Thinking] is to teach them how to think like an economist, a physicist, an artist, and to understand how to use computation to solve their problems, to create, and to discover new questions that can fruitfully be explored.	Hemmendinger (2010)

These definitions tend to focus on the cognitive performances and processes of individuals. Accordingly, we may conclude that activities based on computational thinking are essentially meant to improve cognitive skills and support the processes of teaching and learning in the affected individuals.

Researchers in the field have also held workshops with the aim of establishing the true nature of and a working definition for computational thinking. Some of these workshops have concluded that a rigorous and consistent definition would benefit the field (BİD Workshop Committee, 2011). On the other hand, certain researchers held that attempting to define computational thinking in clear-cut terms is unnecessary and that effort should be applied in establishing the internal relationships within the computational thinking corpus (Voogt et al., 2015: p.726).

> *"There is no clear-cut definition for CT and the main tension in the attempt to define CT has to do with defining the core competencies of CT versus the more peripheral competencies. We argue that for the purpose of conceptualizing CT and integrating it in education, we should not try to give an ultimate definition of CT, but rather try to find similarities and relationships in the discussions about CT (Voget et al., 2015: p.726)."*

Whilst a general concept of computational thinking can be established based on these definitions, they offer little insight into how computational thinking should be applied in practice in the field of education. Practical definitions of computational thinking and its constituents are needed before achievement targets and educational programmes can be created in the classroom. CSTA and ISTE have provided activity rubrics for computational thinking in the years 2011, 2015 and 2016. Table 2 is a list of these activities, sorted according to keywords.

Table 2. Practical computational thinking activities, curated by ISTE.

Keywords	Source
Formulating, organizing, analyzing, modelling, abstractions, algorithmic thinking, automating, efficiency, generalizing, transferring	ISTE (2011)
Creativity, algorithmic thinking, critical thinking, problem solving, cooperation	ISTE (2015; Oden et al. 2015)
Data analysis, abstract thinking, algorithmic thinking, modelling, representing data, breaking problems into components, automation	ISTE (2016) (Computational Thinker Definition)

As these definitions show, the activity lists provide a framework for educators, delineating the educational achievements which they should aim for and outlining methods for assessment and evaluation of these achievements. For example, an educator using these rubrics would know that teaching visual programming tools such as Scratch or KODU in class, is not only meant to help students have fun while designing computer games; They would also use the experience as a medium for instilling some of the concepts and abilities outlined in Table 2.

4. Components of Computational Thinking

The fundamental components of computational thinking are also a source of divergence between researchers. In order to establish a baseline for further analysis, components used by various researchers have been provided in Table 3.

Table 3. Components of Computational Thinking

Components	Source
Abstraction, Algorithms, Automation, Problem Decomposition, Parallelization, Simulation	Barr & Stephenson (2011)
Abstraction, Automation, Analysis	Lee et al. (2011)
Abstraction, Algorithmic Thinking, Decomposition, Evaluation, Generalization	Selby & Woollard (2013)
Abstraction, Algorithms, Decomposition, Debugging, Generalization	Angeli et al. (2016)
Abstraction, Algorithms, Automation, Problem Decomposition, Generalization	Wing (2006, 2008, 2011)

While the exact components may differ, we believe the essential concepts they represent are largely uniform across the field. Computational thinking abilities are essentially the set of skills needed to convert complex, messy, partially defined, real-world problems into a form that a mindless computer can tackle without further assistance from a human (BCS, 2014, p.3). As such, this paper will use the definitions of abstraction, problem decomposition, algorithmic thinking, automation and generalization from amongst the components provided. These definitions can be listed as (Humphreys, 2015):

- *Abstraction* makes problems or systems easier to think about. Abstraction is the process of making an artefact more understandable through reducing the unnecessary detail and number of variables; therefore leading to more straightforward solutions. One of the best-known examples of this is the London Underground example, provided by Humphreys (2015). The London Underground map provides just enough information for the traveller to navigate the underground network without the unnecessary burden of information such as distance and exact geographic position. It is a representation that contains precisely the information necessary to plan a route from one station to another – and no more. Similar examples may be provided for other subjects, allowing the concept to be better understood (Wing, 2008):
    - Verbal and story-based problems in mathematics such as filling rates of pools, areas to be fenced off and accounting calculations are essentially an exercise in abstraction for the students where they are required to separate relevant and irrelevant data and state their solutions in the symbolic language of algebra, geometry, or arithmetic.
    - In geography, students make use of specialized maps (physical, topographic, political, touristic etc.), ignoring many aspects of real-world geography in favour of ease-of-access for data relevant to their current study.
    - History lessons are essentially abstractions of local histories and individual biographies taught as national or world history – abstract projections of real-world events.
- *Problem Decomposition* is a method for taking apart problems and breaking them into smaller and more understandable constituents. This method is also known as "Divide and Conquer".
- *Algorithmic Thinking* is the process of constructing a scheme of ordered steps which may be followed to provide solutions to all constituent problems necessary to solve the original problem.

- *Automation* is the configuration of formed algorithms over computers and technological resources to be efficiently applicable to other problems.
- *Generalization* is the process of adapting formulated solutions or algorithms into different problem states,

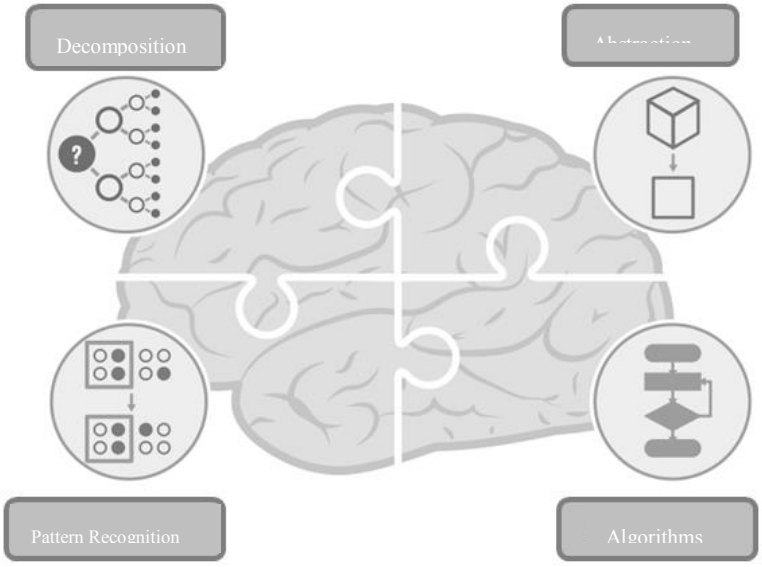

Figure 2. 4 basic strategies for computational thinking (McNicholl, 2018: p.37).

even if the variables involved are different.

There are also a number of techniques used to exemplify and evaluate computational thinking. These comprise the equivalent of a scientific method for computer science. They are employed to put computational thinking to practice in the classroom, at home and at work (Humphreys, 2015):

- Reflection
    - Reflection is the skill of making judgements (evaluation) that are fair and honest in complex situations that are not value-free. Within computer science this evaluation is based on criteria used to specify the product, heuristics (or rules of thumb) and user needs to guide the judgements. A child's realization, when playing with pebbles, that 3 + 4 is the same as 4 + 3 is an example of reflection (or rather, reflective abstraction). The information created in this example is derived not from the pebbles themselves but from the actions taken on them.
- Coding
    - An essential element of the development of any computer system is translating the design into code form and evaluating it to ensure that it functions correctly under all anticipated conditions. Debugging is the systematic application of analysis and evaluation using skills such as testing, tracing, and logical thinking to predict and verify outcomes.
- Designing
    - Designing involves working out the structure, appearance and functionality of artefacts. It involves creating representations of the design, including human readable representations such as flowcharts, storyboards, pseudo-code, systems diagrams, etc. It involves further activities of decomposition, abstraction and algorithm design.

- Analysing
    - Analysing involves breaking down into component parts (decomposition), reducing the unnecessary complexity (abstraction), identifying the processes (algorithms) and seeking commonalities or patterns (generalisation). It involves using logical thinking both to better understand things and to evaluate them as fit for purpose.
- Applying
    - Applying is the adoption of pre-existing solutions to meet the requirements of another context. It is in generalization - the identification of patterns, similarities and connections - and exploiting those features of the structure or function of artefacts. An example includes the development of a subprogram or algorithm in one context that can be re-used in a different context.

5. Critique and Contemporary Research in Computational Thinking

Wing (2006), in the article "Computational Thinking", provided a definition of computational thinking, and held that computational thinking is a fundamental ability for the future which will become a necessity for all individuals and should be employed in the curriculums for students of all levels. However, the article itself in Wing (2006) totaled only 4 pages, was not based on independent research and lacked in-depth analysis of many topics covered in the article. While the article has been used as a foundation for research done by many academics, it has also been put under a heavy amount of critique. Hemmendinger (2010) especially claimed that the components of computational thinking as presented in Wing (2006) are not unique to computational thinking. According to Hemmendiger (2010):

- Reformulating hard problems is typical of all domains of problem solving,
- Philosophers have been thinking about thinking — recursively — for a long time,
- Mathematics surely uses abstraction, and so do all disciplines that build models,
- Separation of concerns and using heuristics also characterizes problem-solving in general.,

Furthermore, Hemmendinger (2010) advances that teaching individuals involved in other disciplines how to think like a computer scientist is unreasonable. Rather than employing a single discipline to dictate the thought processes for all disciplines, physicists should think like physicists and economists should think like economists *while* making use of computational thinking and computational processing technologies in order to solve questions in their field efficiently and determine new questions which would result in novel, efficient methods once solved. Another objection to Wing comes from Denning (2016). According to Denning (2016), the article ascribes an undeservedly significant weight to algorithms and algorithmic thinking. Rather than valuing algorithms above their contribution, Denning (2016) suggests that an algorithmically-controlled computational thinking model should not be ignored as an alternative. Additionally, they advance the notion that computational thinking is not a fundamental skill and cannot be regarded as an equal to fundamental abilities such as reading and writing. In short, the idea that every individual can use computational thinking and campaigns with claims such as "Coding for Everyone", "A Nation of Coders" and "A Coder at Every Home" are unrealistic. The question of whether every profession and every individual really needs to employ computational thinking and consequential coding abilities as a part of computational thinking, is an unresolved discussion in the field. One of the most striking comments on this conundrum is provided by Barr & Stephenson (2011: p.113):

> *The ultimate goal should not be to teach everyone to think like a computer scientist, but rather to teach them to apply these common elements to solve problems and discover new questions that can be explored within and across all disciplines (Barr and Stephenson, 2011: p.113).*

Learning computational thinking and computer science are not one and the same. Yet colloquially, these two expressions are used interchangeably. This supposed equivalency is erroneous as the latter is essentially meant to educate learners in the study and use of the principles of mathematical calculation. One reason why this belief is in wide circulation could possibly be Wing (2006)'s original claim that *"computational thinking is thinking like a computer scientist."*. Denning (2009) and Hemmendinger (2010) oppose this claim mainly because of their thesis that such a definition of computational thinking could give the impression that computational thinking is only relevant to the field of computer science and is largely inapplicable to everyday situations in would-be computational thinking learners.

Programming education is a sub-field of computer science and while primarily conducted to educate learners in the best practices of computer programming, one of its goals is being conducive to the creation of high-quality computer programs. Computational thinking, while it has considerable overlap with computer science on certain elements, focuses mainly on developing and disseminating approaches to problem solving, unlike computer science.

While the terms "coding" and "programming" are used interchangeably with each other, "coding" has been employed as a more exciting and less scary alternative, especially to entice and motivate beginners in scripting. Platforms such as Code Studio, Hour of Code, Code Monkey and MIT's Scratch and App Inventor 2 tend to use coding rather than programming. More advanced text-based and OOP languages (Python, Java etc.) edge towards the use of programming instead. One widely-held belief is that computational thinking, and as a result coding and programming education, has a positive effect on students' problem-solving abilities. Multiple different manifestations of this belief may be observed in contemporary research, and it can be connected to more solid scientific reasoning via analyzing the results of contemporary research:

- Palumbo (1990)'s meta-analysis study concluded that strong evidence to the existence of a meaningful correlation between programming education and problem-solving abilities could not be found. Palumbo (1990) came to this conclusion by evaluating different studies conducted on high school students by a variety of researchers. These included studies based on CAI (Computer Aided Instruction), LOGO and BASIC languages being taught to different groups of students in various class hours and total course length in weeks configurations – none of which discovered a scientifically significant correlation. As previously stated in this article, one of the reasons for the near-extinction of these programming languages may be their inability to provide the expected contribution to the students' problem-solving abilities.
- Kalelioğlu & Gülbahar (2014) held a 5-week long study with 5th Grade Middle School students (22 girls and 27 boys) in the 2013-2014 educational year. Students conducted varying activities in the Scratch programming language as part of the study. Their results indicated that when quantitative data is analyzed, there was no statistically significant divergence between the pre-study and post-study problem-solving ability quotients. Analysis of qualitative data, on the other hand, showed increased student enthusiasm towards programming.
- Kukul & Gökçearslan (2014) worked with 304 5th and 6th grade students who had not taken any programming lessons previously. Similarly, to Kalelioğlu & Gülbahar (2014), they also used Scratch. Their conclusions indicated that no statistically significant change in the students' problem-solving abilities was observed.
- Morelli et al. (2011) analyzed the results under specific indicators. The "App Inventor" mobile programming application was taught to high school students as part of a summer programme. Neither the

"*problem-driven learning*" nor "*support for learning"* indicators mention an increase in the problem-solving abilities of students, instead opting to focus on the increase in motivation observed.

- Wong et al. (2015) conducted an experimental study on 264 5th Grade students in Hong Kong between the years of 2012 and 2014. The first year of the study was used to teach KODU (A game engine developed by Microsoft) to the students, while in the second year Scratch and Small Basic were used in the curriculum. The students' mathematics grade average rose from 74.86 in 2012-2013 to 77.59 in 2013-2014. The students' creativity, critical thinking and problem-solving abilities were also evaluated. Based on t-Test results conducted on data retrieved from the ESDA student evaluation portal, the students' problem-solving abilities appeared to rise from 2.75 to 2.95. However, while the researchers did indicate that participation in coding developed certain abilities in the students, other fundamental abilities were not conclusively affected.

Various strong claims have been made regarding the positive influence of programming/coding education in the cognitive development of children. Papert (1980), believed that programming allowed children to shape their own learning environments. Papert's most important claim was that learning LOGO improved problem-solving abilities by providing concrete experiences which were conductive to conceptualizing pictures on an operational scale (As Papert himself was a mathematician, his examples were frequently based on mathematics and geometry. Concrete experiences were defined as the appearance of geometric shapes on the screen.). Formal operational thinking was defined by Piaget as the ability to construct relationships, make inferences and build hypotheses (Kıncal & Yazgan, 2010: p.724). An individual capable of formal operational thinking can make abstractions, understand mathematical constructs requiring high-level thinking, generalize by applying the acquisitions from these problems to other problems, is able to make plans, and employs a procedural method of thinking. At this point, the similarities between formal operational thinking as defined by Piaget and CT-based abilities become apparent. This is why Papert claimed that LOGO could aid in dismissing negative attitudes towards math in students, teaching mathematical concepts, and strengthening self-control and success-oriented attitudes in children (Liao & Bright, 1991: p.252).

Results from these studies show conflicting opinions in computational thinking literature when it comes to the question of whether programming education on its own has a meaningful effect in the problem-solving abilities of students. But studies where components of computational thinking are employed show an increase in the students' problem-solving, abstract-thinking, troubleshooting and cooperative learning abilities.

- Roman-Gonzales et al. (2017) studied 1251 Spanish students in 5th – 10th grades. CTt (Computational Thinking Test) and PMAt (Primary Mental Abilities Test) were applied to the students. The correlation between CT abilities and "spatial memory", "Reasoning" and "Problem-solving" was calculated experimentally, with spatial memory being k (r=0.44), reasoning (r=0.44) and problem-solving (r=0.67). Problem-solving appears to be more heavily-influenced than other abilities.
- Grover, Pea & Cooper (2015) worked with 54 students in Northern California who were between 11 and 14 years old. A 7-week course was designed for the students where they used the Scratch coding platform and were able to translate their code into text-based platforms based on their acquisitions from the platform. The researchers were able to correlate CT abilities with problem-solving abilities. When the results are analyzed, the students are shown to have advanced themselves especially in algorithmic thinking abilities. Another interesting point is that the students' previous CT experiences and mathematical abilities (as determined by an introductory exam conducted by the researches) were strong

indicators of learning outcomes. Pea & Kurland (1984, p.35) enumerated "mathematical ability", "memory capacity", "analogical reasoning ability", "situational reasoning ability" and "procedural thinking ability" as the mathematical skills necessary for acquisition of programming ability, while also specifying that students who are especially able to operate the LOGO language successfully were also successful in English and humanities classes, and not merely in mathematics.

- Webb (2010) assayed the contribution of programming education to students' troubleshooting abilities. A regimen of 2 hours per week for 5 weeks was planned; with CT skills being connected to problem-solving ability. While 19 boys and 21 girls were present at the beginning, due to personal reasons and exams, only 24 students (16 boys, 8 girls) completed the regimen. At the end of the study, the students were asked to "Fix the Frogger Program" in 40 minutes. Only 1 student failed this assignment, with the rest proceeding to the debugging phase.
- The study conducted by Bers et al. (2013) was based on 3 pre-school classes (2 public and 1 private) of 53 students in total, and had a length of 20 hours. During this study, learners were exposed to 6 main subjects including engineering design processes, robotics, instruction-based programming, loops, sensors, and control mechanisms. TangibleK robots and software were employed in the study. The contents of these subjects were tailored to suit the students' age. Songs, games, and rhythmic and repetitive moves were inserted to the applications. For example, "Simon Says" was used in lesson 3: algorithmic programming and CHERP (Creative Hybrid Environment for Robotic Programming), a drag-and-drop software was taught. The students' troubleshooting, understanding of the relationship between instructions and movement, and use of instruction order and flow-control instructions was studied. The results indicated that students' abilities to cooperate, create ideas, share via negotiation as well as motor skills improved. Furthermore, the students were described to have become more active in their creativity and problem-solving abilities, both in the mathematical and real world.
- The study conducted by Durak and Sarıtepeci (2018) was applied to 156 public school students in Ankara. Two different data collection tools were used in this study. The first one is the personal information form and the second one is the computational thinking ability form. In this study, the factors affecting the computational thinking skills of students were examined. These factors are gender, education level, IT usage experience, daily internet usage period, mathematics achievement, attitudes towards the mathematics course, attitudes towards science courses, achievements in science courses, achievements in information technology courses and attitudes towards information technology courses. Among these factors, it was determined that the most effective factors on computational skills were education level, mathematics achievement, attitude towards the mathematics course and attitude towards science courses.

Upon analysis of these studies, it becomes apparent that it is lessons in coding, mathematics, natural sciences, social sciences and language arts, taught according to computational thinking skills and not mere programming or coding education, which affect an increase in the problem-solving, abstract thinking, troubleshooting, procedural thinking and similar abilities in students. An appropriate and interdisciplinary application of the component of CT abilities needs to be advanced in order to raise students not only as coders but as individuals with a radical way of thought and perspective. Furthermore, it may be appropriate for Computational thinking and STEAM (Science, Technology, Engineering, Arts and Mathematics) to be considered together as these two fields share a great deal of subject material (Gülbahar, 2017: p.331). Interdisciplinary work on the part of the students and their ability to realize relationships between areas of study, determine the problems they are facing, investigate potential solutions, decide upon the correct solution, gather data, analyze data, troubleshoot, develop their models and generalize

solutions (ISTE, 2016) will aid their problem-solving abilities.

6. Conclusion

Computer science-based technologies are developing rapidly in our era, influencing the problem-solving processes and social lives of both individuals and societies. From medical work to social media use, results of computer science studies are integrated to the daily lives of individuals in a multitude of fields. The effects of computer science on modern society is also an indicator of its effects on the scientific method and therefore, naturally, scientists. Natural scientists have long positioned computation as a "third" foundation of the scientific method alongside theory and experimentation, and that computational thinking is essential to their work (Denning, 2009).

Though the definitions of and framework for computational thinking as set out by Wing (2006) have long been critiqued by other researchers, the importance of computer science has been growing daily, finding applications in multiple fields from curing disease to preventing terrorist attacks. Nonetheless, the claim that computer science and as a results computational thinking is a fundamental discipline on par with reading, writing and basic arithmetic, is still being debated.

Populist notions such as "Computer Science and Computational Thinking for All", aimed at bringing the field to the mainstream, will make it more difficult for the field to preserve its rightful rigour. As we have deducted from the works of Denning, Hemmendinger and Barr amongst others presented in this article, ascribing an undeserved importance to certain fields – whether they be deemed coding, computer science, or computational thinking – would be inappropriate. Still, researchers may benefit from holding computational thinking as a potential method of transforming education, as long as they also hold the criticisms applied to the field in equal regard. As Denning (2010, p.28) has also stated, holding computational thinking (and coding) in (undeservedly) excessive esteem may lead us back to the same pitfalls we are attempting to avoid.

As a final remark, we hold that the fundamental goal of computational thinking (and instilling this ability in students) and computer education should be aiding students in understanding and – through use of their creative impulses – changing the world they live in, for the better (Department for Education, 2014, p.217).

References

Agalianos, A., Noss, R., & Whitty, G. (2001). Logo in mainstream schools: the struggle over the soul of an educational innovation. *British Journal of Sociology of Education*, *22*(4), 479-500.

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 computational thinking curriculum framework: Implications for teacher knowledge. *Journal of Educational Technology & Society*, *19*(3), 47.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community?. *Acm Inroads*, *2*(1), 48-54.

BCS, The Chartered Institute for IT. 2014. Call for evidence - UK Digital Skills Taskforce. http://bit.ly/ILi8mdn [Retrieved 17.01.2018].

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education*, *72*, 145-157.

Bringuier, J. C. (1980). Conversations with Jean Piaget. *Society*, *17*(3), 56-61.

Çınar, M. & Tüzün, H. (2017, February). Bilgisayımsal Düşünme Sürecinin Doğasına İlişkin Nitel Bir Analiz (A

Qualitative Analysis on the Nature of the Computational Thinking Process). Presented to 19. Akademik Bilişim Konferası (Conference on Academic Informatics), Aksaray University, retrieved 24.12.2017 from http://ab.org.tr/ab17/ozet/233.html.

Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?. *Computers & Education*, *58*(1), 240-249.

Denning, P. J. (2009). The profession of IT Beyond computational thinking. *Communications of the ACM*, *52*(6), 28-30.

Department for Education. 2014. The National Curriculum in England, Framework Document. Reference: DFE-00177-2013. Retrieved 26.12.2017 from: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/335116/Master_final_national_curriculum_220714.pdf.

Durak, H. Y., & Saritepeci, M. (2018). Analysis of the relation between computational thinking skills and various variables with the structural equation model. Computers & Education, 116, 191-202.

Etherington,C. (2017), Retrieved 24.12.2017 from: https://news.elearninginside.com/how-plato-changed-the-world-in-1960/.

Furber S (2012) Shut down or restart? The way forward for computing in UK schools. Technical report, The Royal Society, London.

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, *25*(2), 199-237.

Hemmendinger, D. (2010). A plea for modesty. *Acm Inroads*, *1*(2), 4-7.

Humpreys, S. (2015). Computational Thinking, a guide for teacher. Computing at School. Charlote BCS. The Chartered Institue for IT

ISTE (2011), Operational definitions of computational thinking, retrieved 24.12.2017 from: https://c.ymcdn.com/sites/www.csteachers.org/resource/resmgr/CompThinkingFlyer.pdf.

ISTE (2016), ISTE Standarts for Students, retrieved 24.12.2017 from: http://www.iste.org/docs/Standards-Resources/iste-standards_students-2016_one-sheet_final.pdf?sfvrsn=0.23432948779836327.

Kalelioglu, F., & Gülbahar, Y. (2014). The effects of teaching programming via Scratch on problem solving skills: a discussion from learners' perspective. *Informatics in Education*, *13*(1), 33.

Kalelioglu, F., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing*, *4*(3), 583.

Keser, H. (2011). Türkiye'de Bilgisayar Eğitiminde İlk Adım: Orta Öğretimde Bilgisayar Eğitimi İhtisas Komisyonu Raporu (Turkey's First Steps in Computer Education: Specialized Commission on Computer Education in Secondary Education Report). *Eğitim Teknolojisi Kuram ve Uygulama (Theoretical and Practical Educational Technologies)*, *1*(2), 83-94.

Kıncal, R. Y., & Yazgan, A. D. (2010). Investigating the formal operational thinking skills of 7th and 8th grade primary school students according to some variables. *Elementary Education Online*, *9*(2), 723-733.

Korkmaz, Ö., Çakır, R., Özden, M. Y., Oluk, A., & Sarıoğlu, S. (2015). Bireylerin Bilgisayarca Düşünme Becerilerinin Farklı Değişkenler Açısından İncelenmesi (A Multi-Variable Investigation of the Computational Thinking Abilities of Individuals). *Ondokuz Mayıs Üniversitesi Eğitim Fakültesi Dergisi (19th May University*

Faculty of Education Journal), *34*(2), 68-87.

Korkmaz, Ö., Çakır, R., Özden, M. Y., Oluk, A., & Sarıoğlu, S. (2015). Bireylerin Bilgisayarca Düşünme Becerilerinin Farklı Değişkenler Açısından İncelenmesi (A Multi-Variable Investigation of the Computational Thinking Abilities of Individuals). *Ondokuz Mayıs Üniversitesi Eğitim Fakültesi Dergisi (19th May University Faculty of Education Journal)*, *34*(2), 68-87.

Kukul, V., & Gökçearslan, Ş. (2014). Scratch ile programlama eğitimi alan öğrencilerin problem çözme becerilerinin incelenmesi. (Investigation of the Problem-solving Skills of Students with Scratch-based Programming Education.)

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

Liao, Y. K. C., & Bright, G. W. (1991). Effects of computer programming on cognitive outcomes: A meta-analysis. *Journal of Educational Computing Research*, *7*(3), 251-268.

Logo Foundation (2015). Logo and Learning, retrieved 24.12.2017 from: http://el.media.mit.edu/logo-foundation/what_is_logo/logo_and_learning.html.

McNicholl, R.(2018). Computational thinking using code.org. Hello World, 4, 37.

Morelli, R., De Lanerolle, T., Lake, P., Limardo, N., Tamotsu, E., & Uche, C. (2011, March). Can android app inventor bring computational thinking to k-12. In *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)* (s. 1-6).

National Research Council. (2010). Committee for the Workshops on Computational Thinking. In *Report of a workshop on the scope and nature of computational thinking, Natl Academy Pr*.

Palumbo, D. B. (1990). Programming language/problem-solving research: A review of relevant issues. *Review of educational research*, *60*(1), 65-89.

Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism*, *36*(2), 1-11.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New ideas in psychology*, *2*(2), 137-168.

Román-González, M., Pérez-González, J. C., & Jiménez-Fernández, C. (2016). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. Computers in Human Behavior, 1-14

Selby, C., & Woollard, J. (2013). Computational thinking: the developing definition.

Tekerek, M., & Altan, T. (2014). The effect of scratch environment on student's achievement in teaching algorithm. *World Journal on Educational Technology*, *6*(2), 132-138.

Voogt, J., Fisser, P., Good, J., Mishra, P., & Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, *20*(4), 715-728.

Wing, J. (2014). Computational thinking benefits society. *40th Anniversary Blog of Social Issues in Computing*, *2014*.

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

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical transactions of the royal society of London A: mathematical, physical and engineering sciences*, *366*(1881), 3717-3725.

Wing, J.M. (2011), Research Notebook: Computational thinking -what and why? The Link Magazine, 20-23. https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

Yadav, A., Mayfield, C., Zhou, N., Hambrusch, S., & Korb, J. T. (2014). Computational thinking in elementary and secondary teacher education. *ACM Transactions on Computing Education (TOCE)*, *14*(1), 5.

Yecan, E., Özçınar, H., & Tanyeri, T. (2017). Bilişim Teknolojileri Öğretmenlerinin Görsel Programlama Öğretimi Deneyimleri (A Collection of Visual Programming Experiences by Information Technologies Educators). *İlköğretim Online (Elementary Education Online)*, *16*(1).

IJCSES

Volume 3, No:1

April 2019

ISSN 2513-8359