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

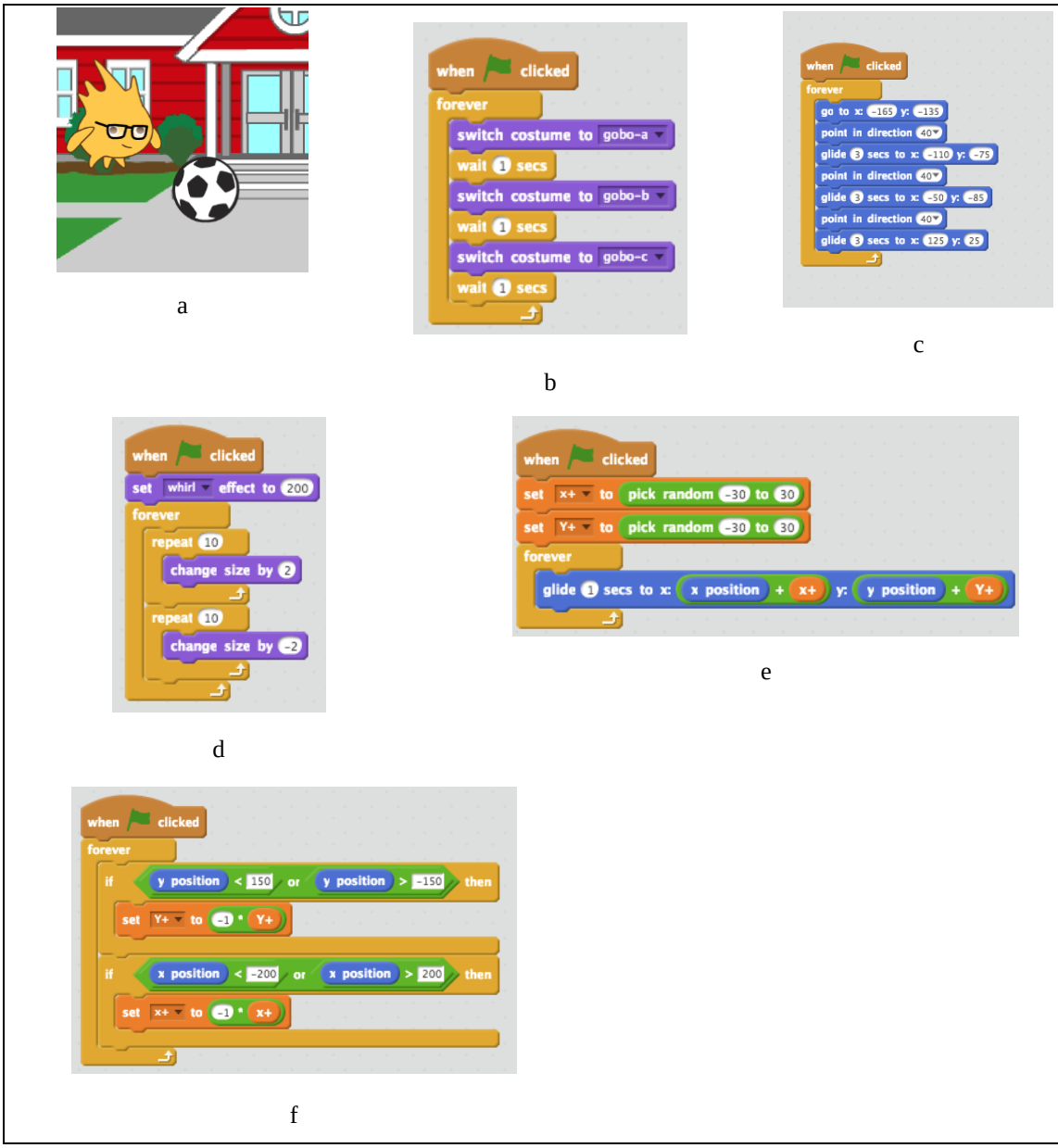


Figure 1. Code blocks of week 1

*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<br>• Moving sprites according to keystrokes<br>• Resetting the game when being hit by objectives<br>• Resetting the game when touching the maze<br>• Ending a game when arrive at the finish line |
| Program development | Using forever and if blocks to create the event handlers<br>• with arrow keys<br>• 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.

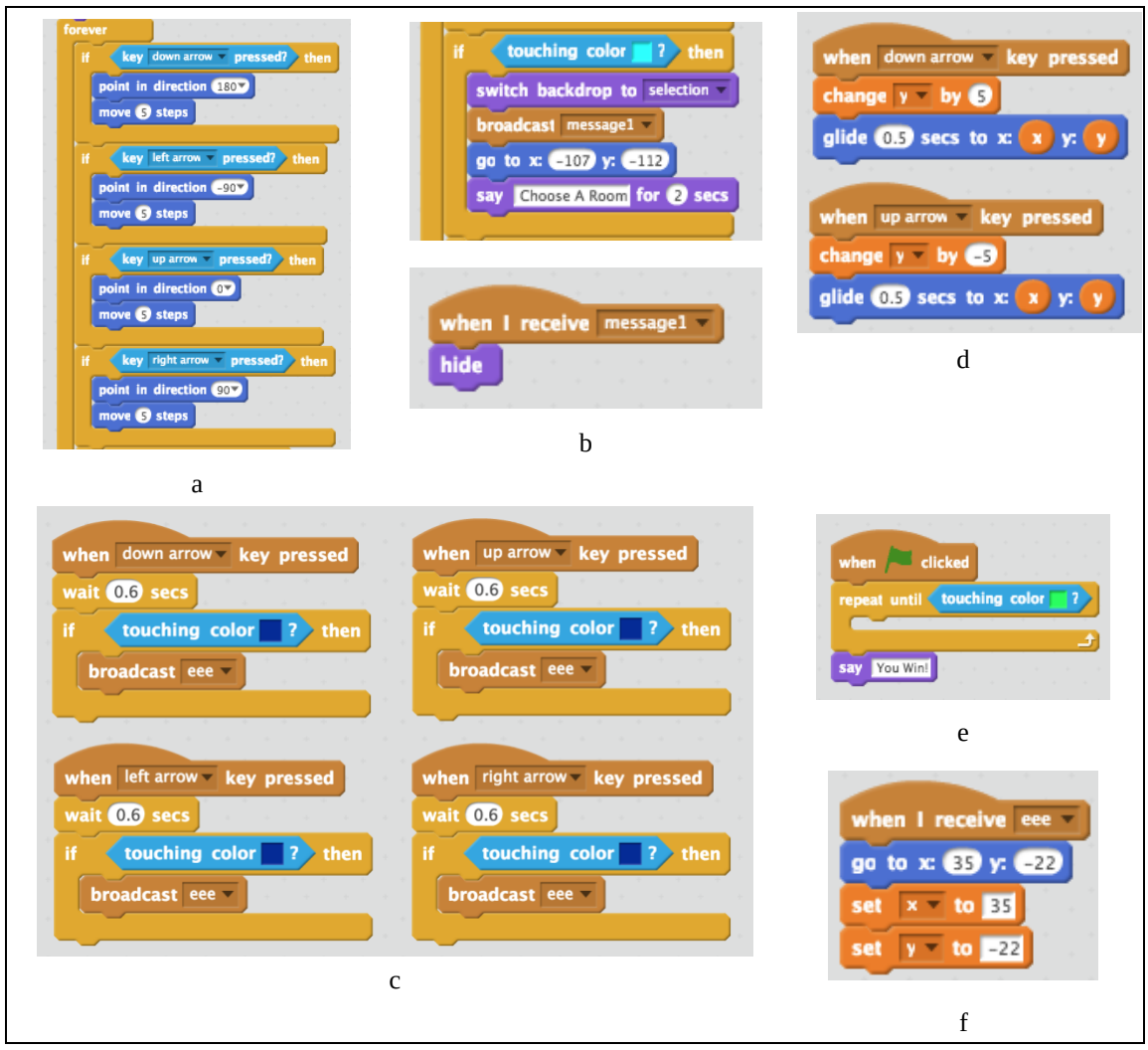


Figure 2. Code blocks of week 2

*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)<br>• Gaining or losing a score when accomplish or fail a mission<br>• Moving sprites according to keystrokes<br>• Showing/hiding objects in random locations<br>• Finishing game according to the score |
| Program development | • Using variables to save and update values<br>• Using forever and if blocks to create the event handlers<br>  o with arrow keys<br>  o with touch<br>• Using random block to display objects in random places<br>• 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.

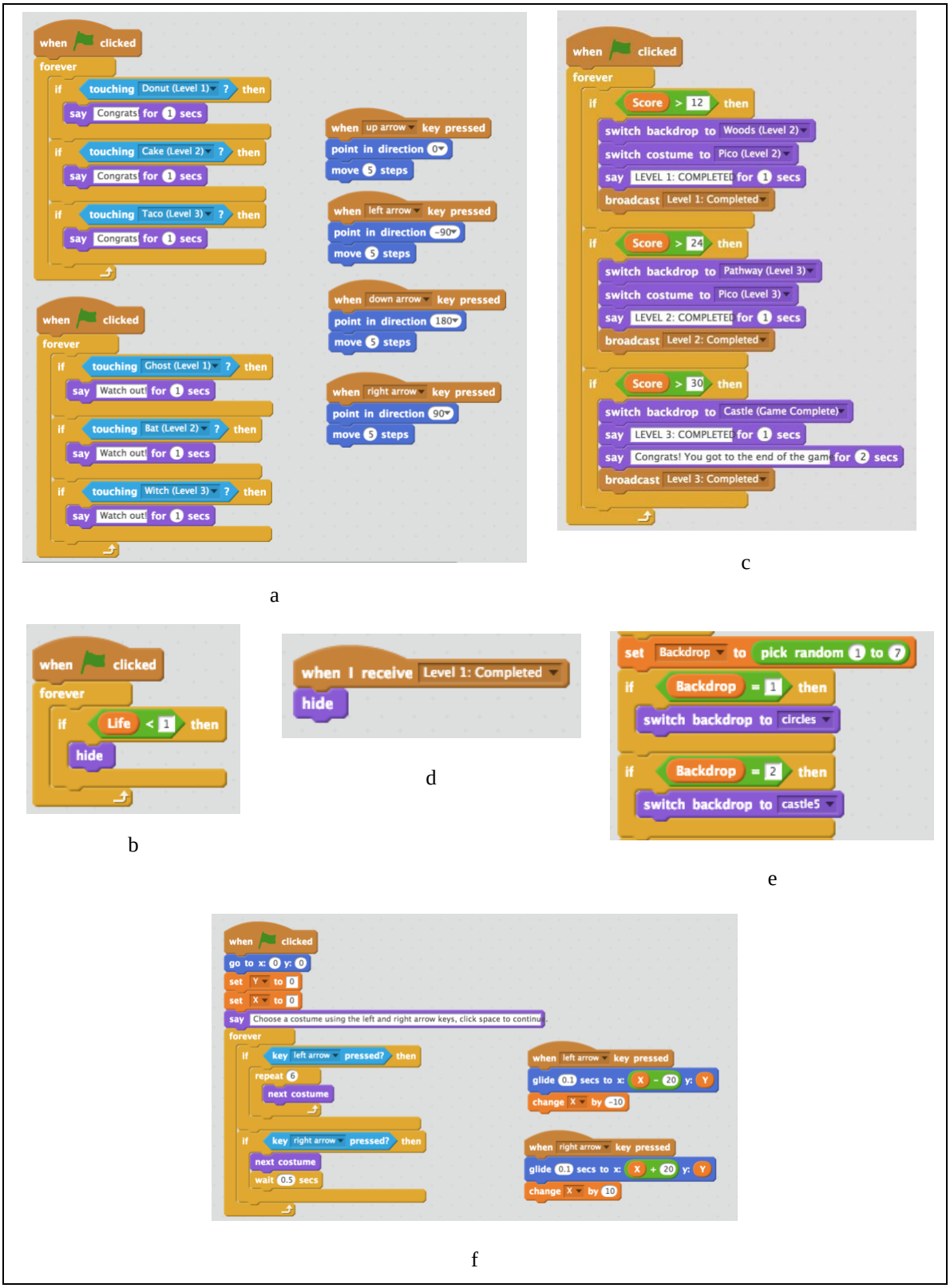


Figure 3. Code blocks of week 3

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<br>• Checking answers entered by users<br>• Providing feedback according to the answers |
| Program development | • Using ask block to show questions<br>• Using if-else block to compare user inputs and correct answers<br>• 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.

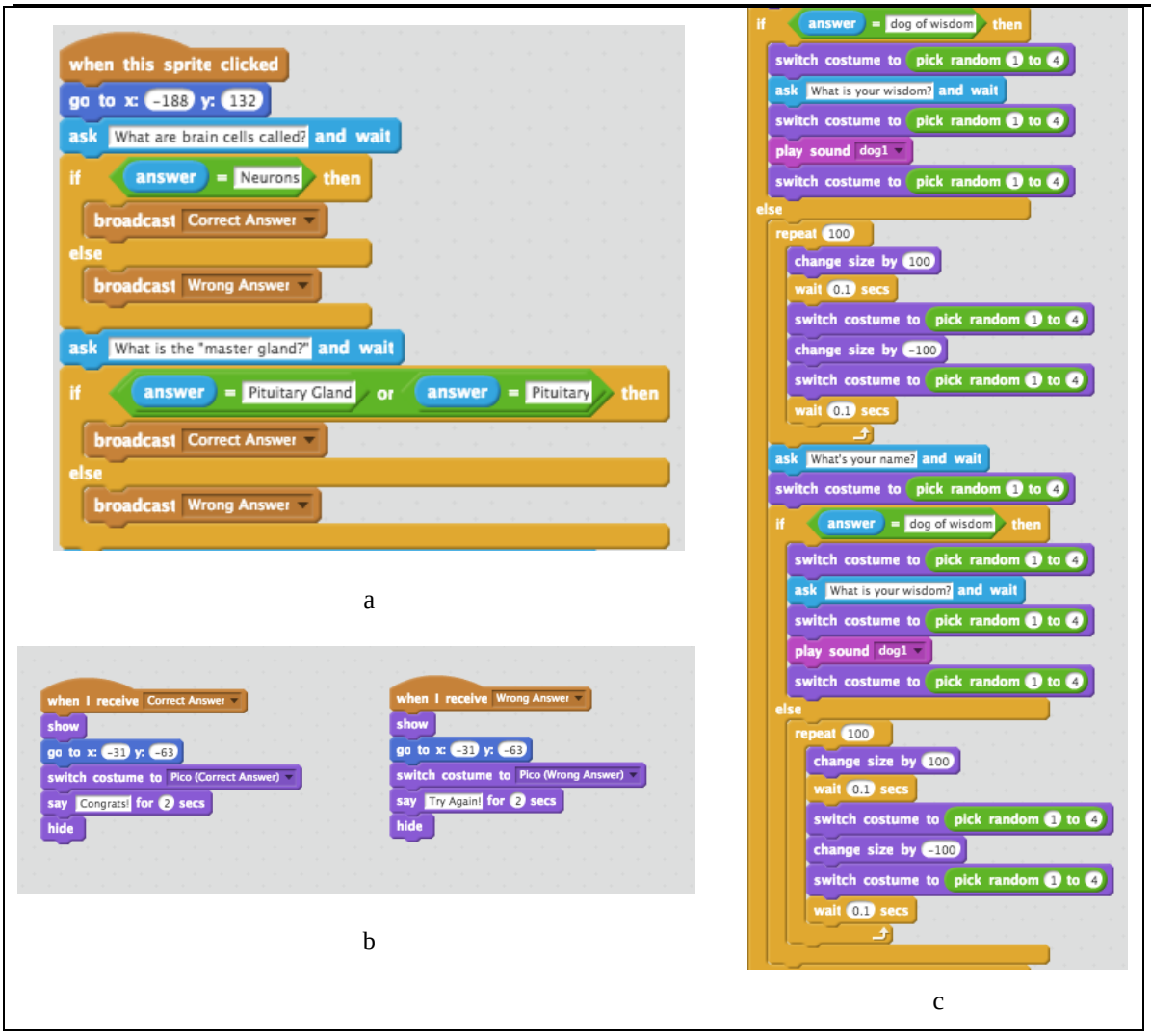


Figure 4. Code blocks of week 4

## 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