


Abstractive-Based Programming Approach to Computational Thinking: Discover, Extract, Create, and Assemble

Journal of Educational Computing Research
2022, Vol. 0(0) 1–34
© The Author(s) 2022
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/07356331221134423
journals.sagepub.com/home/jec


Ndudi O. Ezeamuzie 

Abstract

Most studies suggest that students develop computational thinking (CT) through learning programming. However, when the target of CT is decoupled from programming, emerging evidence challenges the assertion of CT transferability from programming. In this study, CT was operationalized in everyday problem-solving contexts in a learning experiment ($n = 59$) that investigated whether learning programming enhances students' CT skills. Specifically, this study examined the influence of a novel, systematic and micro instructional strategy that is grounded in abstraction and comprised of four independent but related processes – discover, extract, create, and assemble (DECA) towards simplification of problem-solving. Subsidiary questions explored the effects of students' age, gender, computer proficiency, and prior programming experience on the development of CT. No significant difference was found between the CT skill and programming knowledge of the groups at the posttest. However, within-group paired t-tests showed that the experimental group that integrated DECA had significant improvement in CT but not in the control group across the pretest-posttest axis. Implications of the inconclusive finding about the transfer of programming skills to CT are emphasized and the arguments for disentangling CT from programming are highlighted.

Faculty of Education, University of Hong Kong, Hong Kong

Corresponding Author:

Ndudi O. Ezeamuzie, Faculty of Education, University of Hong Kong, RunMe Shaw Building, Hong Kong.
Email: amuzie@connect.hku.hk

Keywords

computational thinking, problem-solving, programming, abstraction, constructionism

Introduction

Wing (2006) described computational thinking (CT) as ‘thinking like a computer scientist’ that everyone can harness to solve problems (p. 34). Understandably, the claim that CT enhances problem-solving is a goal that resonates with every educator, which Jonassen (2000) described as ‘the most important learning outcome for life’ (p. 63). To achieve the goal of raising CT problem solvers and developing technology-aware citizens, one literacy that has dominated the learning ecosystem is computer programming (Ezeamuzie & Leung, 2022). Although CT was explicitly distinguished as ‘conceptualizing, not programming’ (Wing, 2006, p. 35), the reviews on CT practices showed that programming remains the primary approach for developing and assessing CT (Ezeamuzie & Leung, 2022; Lye & Koh, 2014).

Whereas Wing’s (2006) comparison of CT to time-honoured literacies of reading, writing, and arithmetic is open to scholarly debates, several positive outcomes have been reported in CT and programming empirical studies. In a meta-analysis (Scherer et al., 2019), learning programming enhances students’ CT and cognitive skills such as reasoning, creativity, metacognition, mathematical thinking, and natural language literacy. Other benefits of CT include enhancement of social skills, self-management, collaboration, communication, and confidence (Denner et al., 2019; Popat & Starkey, 2019). These benefits spread across all levels of learners including early childhood education (Bers et al., 2014).

However, the heightened benefits of learning CT through programming (e.g., Denner et al., 2019; Popat & Starkey, 2019; Scherer et al., 2019) contrast with the antecedents of studies on the cognitive benefits of programming. Impressions that learning programming enhances generic and transferrable problem-solving skills are unsubstantiated (Guzdial, 2015; Pea & Kurland, 1984). According to Denning (2017), there is no evidence to support the claim of learning transfer in programming. These disparities evoke questions about what has changed in programming. Does learning to program augment the development of CT skills?

To gain clarity on this question, this study reports an experiment in a middle school classroom that investigated the effects of abstractive-based programming – an instructional strategy that mirrors the inherent nature of abstraction. The abstractive-based model consists of four independent but connected activities – *discover*, *extract*, *create*, and *assemble* (DECA) depicting various ways abstraction has been implemented in CT (Ezeamuzie, Leung, & Ting, 2022). By adopting an instructional strategy that originated from the inherent nature of CT, the hypothesis guiding this asserts that learning programming with explicit DECA instruction enhances the transfer of skills to CT.

Background

This section establishes the meaning of CT and overviews the literature to elucidate the influence of programming on learners' cognitive skills and CT. Specifically, the effects of the diverse modelling of CT assessments are analysed. The theoretical framework is expounded to illustrate DECA, an abstractive-based programming strategy and the operationalization of CT through Programme for International Student Assessment (PISA). Finally, the research questions are highlighted explicitly.

Computational Thinking

The call for CT emerged from the assumption that the ways computer scientists think are transferable in solving problems beyond the boundaries of computer science (Wing, 2006). While Wing (2006) was a significant moment in the CT discussion, other accounts exist from historical lens (Tedre & Denning, 2016). Notably, Alan Perils' call for programming in the liberal arts, (Guzdial, 2008), Donald Knuth's view of how programming strengthens conceptual clarity (Knuth, 1974) and Seymour Papert's idea of learning by doing when children program (Papert, 1980). Some studies have criticized Wing's (2006) framing of CT as overly broad (Mannila et al., 2014) and many frameworks have been proposed to conceptualize CT. These include CT conceptual models by Computing at School (Csizmadia et al., 2015), International Society for Technology in Education (Barr et al., 2011; Barr & Stephenson, 2011), and others (e.g. Brennan & Resnick, 2012; Korkmaz et al., 2017; Selby & Woollard, 2013; Shute et al., 2017; Weintrop et al., 2016).

There is neither consensus nor shortage of interpretations for CT. While a consensus model of CT may be desirable for learning and assessment, Nardelli (2019) advised against such mapping and suggested that CT should be interpreted 'as a shorthand' of computer science for all students (p. 32). A systematic review of literature by Ezeamuzie and Leung (2022) found that most studies that assessed the development of CT implemented CT as the composite of programming concepts and preferred models that originated from assessment instruments. Moreover, a substantial number of CT studies had no clear difference between CT and programming (Ezeamuzie & Leung, 2022; Lye & Koh, 2014). To avoid narrow views about CT (Denning et al., 2017) and exaggerated claims about CT (Tedre & Denning, 2016), this study adopts the meaning of CT from Wing's (2006) description of CT as thinking like a computer scientist but decontextualized from programming.

Programming and Transfer of Cognitive Skills

Arguments about learning programming and the transfer of problem-solving to other domains are inconsistent. For example, in the 20th century, Pea and Kurland (1984) examined the impact of learning programming on learners' development of higher-order cognitive skills and found that existing evidence failed to support the claims of

transfer. However, a meta-analysis by [Liao and Bright \(1991\)](#), comprising 432 effect sizes in 65 studies showed that learning programming improves students' cognitive ability moderately ($d = .41$). The updated meta-analysis comprising 86 effect sizes in 22 studies found a strong transfer effect ($d = .76$) between programming and learners' cognitive abilities ([Liao, 2000](#)).

The inconclusive nature of transfer effects has persisted across the 21st century. According to [Guzdial \(2015\)](#), the assertion that learning to program will improve generic problem-solving skills has not been substantiated. In examining the contentious issues on CT, [Denning \(2017\)](#) advised against blind acceptance that learning to program will enhance learners' CT problem-solving abilities and challenged educators to empirically verify the unsubstantiated claims. In contrast, some studies asserted that learning to program enhances CT. For example, in a meta-analysis on the transfer effect of programming on cognitive skills, [Scherer et al. \(2019\)](#) analysed 105 interventions with 539 effect sizes. They separated the cognitive outcomes into near-transfer and far-transfer; a classification that reflects in finer granularity the distance of the cognitive skills from programming. Near-transfer denoted the class of outcomes when programming skills were assessed. Far-transfer represented other cognitive outcomes such as reasoning, mathematical skills, spatial skills, metacognition, and creativity. [Scherer et al. \(2019\)](#) found a strong and moderate effects in near transfer ($g = .75$) and far transfer ($g = .47$), respectively.

Based on the findings of their meta-analysis, [Scherer et al. \(2019\)](#) asserted that "the overall claims that programming aids other forms of thinking skill can therefore be substantiated" (p. 783). Though an interesting finding in the prevailing arguments of transfer, [Scherer et al.'s \(2019\)](#) classification of CT as a near transfer skill (i.e., part of the programming knowledge) instead of the far-transfer, contradicts [Wing's \(2006\)](#) position on CT as 'conceptualizing, not programming' (p. 35). This contradiction is consistent with the findings in [Ezeamuzie and Leung \(2022\)](#) that many empirical studies did not distinguish between CT and programming. Whereas CT is rooted in computer science concepts (including programming concepts), the CT skills expounded in [Wing \(2006\)](#) entail using computer science concepts to solve problems that are decontextualized from programming.

Transfer of Computational Thinking Confounded by Nature of Assessment

The close association between programming and CT assessment is evident from the instruments that researchers have adopted in empirical studies ([Ezeamuzie & Leung, 2022](#)). For instance, in [Merkouris et al. \(2017\)](#), secondary school students' ability to read and understand program codes was the underlying skill in the assessment of the influence of robotics and wearables on CT. Also, CT assessment instruments such as Dr Scratch ([Moreno-León et al., 2015](#)) and CT-test ([Román-González et al., 2017](#)) are interweaved in programming. Dr Scratch assigns CT scores by counting programming features in Scratch projects, which reflect students' code writing ability. Similarly, CT-test consists of 28 multiple choice questions designed as code.org program snippets for

testing knowledge of sequence, loops, conditionals, and functions through code reading. Hence, studies that measured CT with Dr Scratch (e.g., [Wei et al., 2021](#); [Zhao et al., 2022](#)) and CT-test (e.g., [Pérez-Marín et al., 2020](#); [Taylor & Baek, 2019](#)) are rooted in programming knowledge.

Another assessment approach is self-reports. Technically, the self-reports measure CT disposition rather than skill development. The CT-scale by [Korkmaz et al. \(2017\)](#) is an example of a self-reporting instrument that measures CT disposition by aggregating scores in five dimensions – creativity, algorithmic thinking, critical thinking, problem-solving, and cooperativity. Studies that adopted CT-scale (e.g., [Lee & Lee, 2021](#); [Saritepeci, 2020](#)) require learners to select their disposition from a 5-point Likert scale ranging from never (1) to always (5). Items such as “I like the people who are sure of most of their decisions” ([Korkmaz et al., 2017](#), p. 565) demonstrate dispositions and beliefs.

Besides programming-enabled instruments and self-reports, other forms of CT assessments illustrate how CT is detached from programming. In Bebras, each task is designed to measure one or more of the sub-CT skills of abstraction, algorithms, decomposition, evaluation, and generalisation ([Dagienė & Sentance, 2016](#)). Using Bebras, neither code writing nor code reading was required in assessing the influence of robotics programming on primary schoolers’ CT skills ([Baek et al., 2019](#); [Noh & Lee, 2020](#)). Modelling CT assessments after real-life scenarios to measure students’ ability to transfer algorithmic and logical flow to dissimilar contexts is another approach. For instance, in assessing CT, [Witherspoon et al. \(2017\)](#) probed students’ understanding of autonomous car designs and the application of sensors in tracking vehicular capacity on a bridge after a robotic programming intervention. Other assessments combined both programming tasks with everyday problem-solving tasks ([Chen et al., 2017](#); [Shen et al., 2020](#)).

The multi-faceted approach to CT assessment demands that outcomes of studies should be interpreted with caution. The highlighted differences in assessments suggest that significant outcomes in empirical interventions are not sufficient; it is equally important to verify what was measured. In this study, CT is decoupled from programming and aligned with everyday problems in the well-validated PISA instrument to gain clarification on whether learning programming enhances CT skills.

Programme for International Student Assessment: Operationalizing Computational Thinking

The Programme for International Student Assessment (PISA) by the Organisation for Economic Co-operation and Development ([OECD, 2013a](#)) is adopted for assessing CT ability in this study. PISA looks beyond school subject knowledge to measure the ability of 15-year-old students to apply their knowledge and skill in solving real-life problems. Undeniably, what constitutes problem-solving is complex and amorphous, especially when loosely associated with a wide variety of tasks in education. Although it is theoretically difficult to frame the dimensions of problem-solving, several models

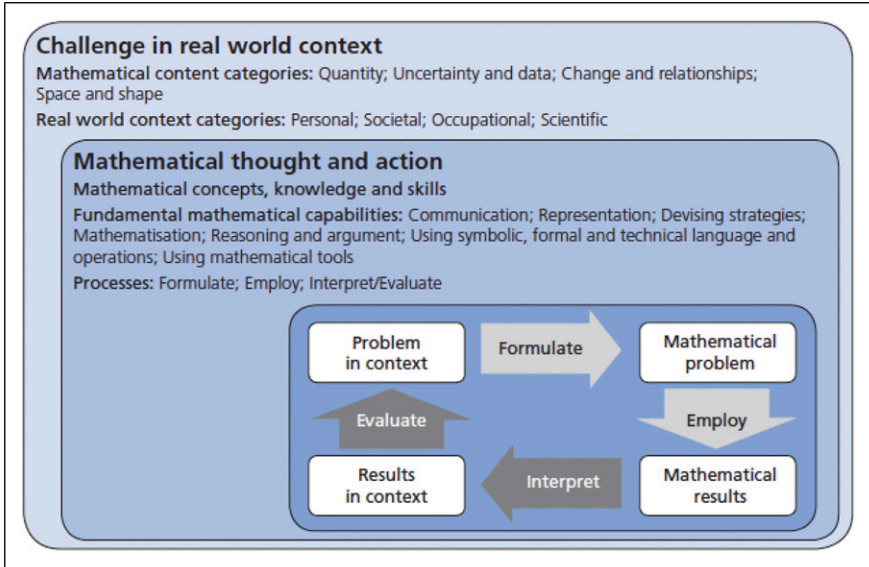


Figure 1. Problem-solving in the PISA 2012 mathematical literacy framework. This figure is reproduced from (OECD, 2013b).

provide insight into the nature of problems. For example, problems may be classified into well-structured or ill-structured (Jonassen, 1997) and routine or non-routine (Mayer, 1998). To account for the shortfalls of these dichotomous classifications, Jonassen (2000) modelled problems in terms of structuredness, complexity, and domain specificity. Concerning CT, everyday problems mask the implementation of CT practices; requiring intentional restructuring of problems to make the application of CT explicit (Ezeamuzie, Leung, Garcia, et al., 2022). PISA was adopted because it focuses on what students do with their knowledge in solving everyday problems, which is consistent with the purpose of CT – a problem-solving skill (Wing, 2006).

Established in 1997, the PISA survey is a triennial programme that measures students' abilities in three major domains of reading, mathematics, and science. With the first edition in 2000, PISA 2012 was the fifth cycle of PISA and was conducted in 66 countries/economies with mathematical literacy as the focal domain (OECD, 2013a). Mathematical literacy represents an individual's capacity to use mathematical concepts, procedures, facts, and tools to describe, explain and predict phenomena. By reasoning mathematically, individuals make well-founded judgments and decisions (OECD, 2013b). Succinctly, mathematics literacy is distinguished from isolated knowledge of mathematics concepts but emphasizes problem-solving that applies domain-based knowledge in real-life challenges. Figure 1 shows the mathematical literacy framework situated in real-life problem-solving contexts and framed under three dimensions: *content, process, and context*

Content dimension of the framework represents the mathematical knowledge domain of the problem scenario. PISA 2012 adopted the historical structure of mathematics since the 17th century in four broad areas – change and relationship, quantity, space and shape, and uncertainty and data. *Process* dimension sums up the mathematical actions that are required to solve problems in diverse contexts in three approaches – formulating, employing, and interpreting. Formulating provides structure by translating the problems from the semantics of the real world to a mathematical structure, expressions, or equations. Employing entails solving problems by applying arithmetic computations and solving equations. Interpreting involves evaluating mathematical arguments to make a reasoned inference about problems. *Context* dimension describes the domain where the problem is situated. While this list of contexts is inexhaustive, PISA 2012 identified four areas of problem-solving. Occupational context depicts the category of problems that people encounter at work. Personal context refers to issues and challenges that affect individuals, homes, and relatives. Societal context identifies problems that are situated in local and global communities such as government services and educational policies. Scientific context focuses on science and technological issues of nature such as weather change, genetics, and biodiversity.

Abstractive-based Instruction: Discover, Extract, Create, and Assemble (DECA)

In this study, an instructional strategy that mirrors the inherent nature of abstraction in CT is adopted for examining the effects of programming on CT. Abstraction is the root of breakthroughs in computer science (Colburn & Shute, 2007; Kramer, 2007) and a regular dimension in most CT frameworks. Analysis of the operational models of abstraction revealed four independent but connected activities – *discover*, *extract*, *create*, and *assemble* (DECA) that depicts various ways of implementing abstraction in CT (Ezeamuzie et al., 2022). Figure 2 shows that abstraction is not restricted to specific set of activities but the summation of cognitive activities with the primary goal of simplifying problem-solving. In the DECA abstractive-based model, students reflect on the programming problems to understand the applicable solutions. *Discover* examines entities or the operation of a process to uncover underlying patterns. *Extract* focuses on distinguishing between irrelevant, redundant, and relevant features. *Create* synthesizes generic classification rules based on the pattern for automation. *Assemble* combines the relevant feature and common pattern systematically to form a functional unit.

Research Question

RQ 1. Does abstractive-based learning of programming enhance students' CT?

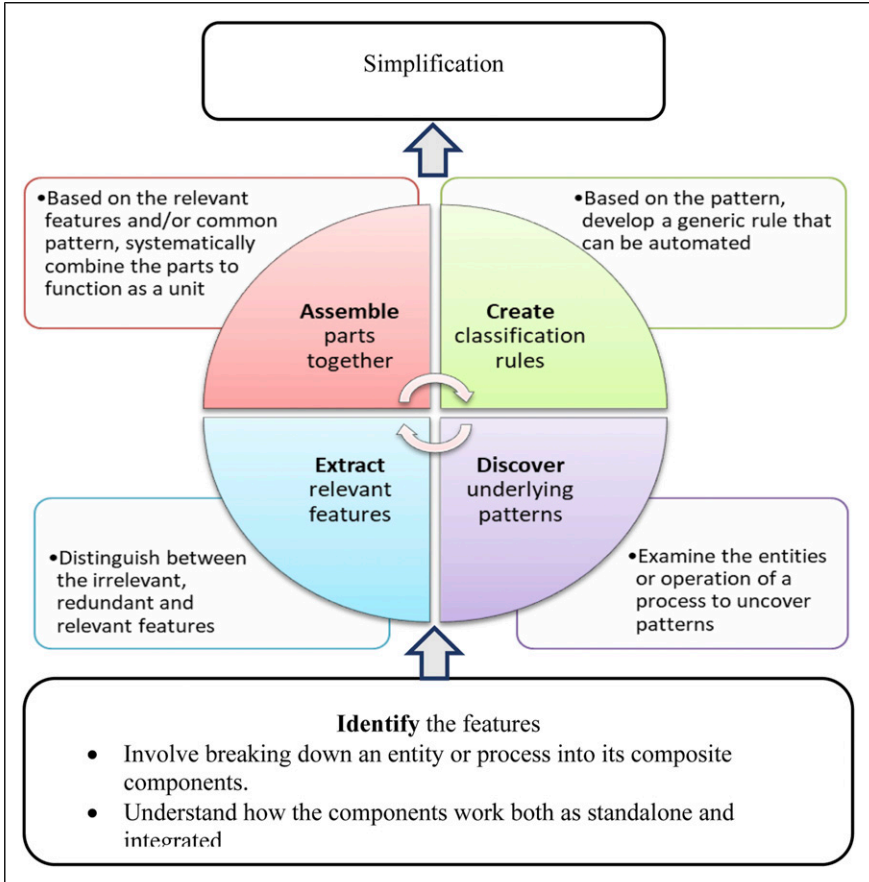


Figure 2. Summary of the core activities and purposes of abstraction (Ezeamuzie et al., 2022).

RQ 2. Does abstractive-based learning of programming enhance students' programming knowledge?

RQ 3. Are there any interaction effects of abstractive-based programming and students' characteristics (age, computer proficiency, prior programming experience, gender) towards CT and programming knowledge?

Method

A quantitative experimental design was adopted in this study. Participants were selected and assigned to the experimental and control groups, randomly. Data were collected at various stages of the experiment through questionnaires and tests.

Participants

An invitation to join the study was sent to five secondary schools in a suburban city in West Africa. These schools were selected because CT research in developing countries is scarce and programming education is hugely missing in their classrooms (Ezeamuzie, 2022). By situating this investigation among participants from unrepresented regions, this study seeks to uncover peculiar prospects and challenges in technology-deprived schools. Schools were deemed eligible to join this intervention if they accepted the invitation and have a functional computer lab with minimum of 30 computers that are provisioned to support online classes. While the schools expressed interest to join the learning intervention, they were unable to meet the requirements. One of the secondary schools with 31 computers was funded to set up an internet connection and stable power supply throughout the 10-week learning period. The selected school runs co-educational training and has a total student population of about 1200 students across the six grades. The 180 science students in the fourth and fifth years of secondary school (equivalent to grades 10 and 11 in the K–12 system) were invited to join this study before their long-term holiday (equivalent to the summer holiday). As approved by the institutional review board, students and parental consent were obtained before the study. Table 1 shows the summary of the participants. More than 70% of the participants in this study had no prior programming experience. Also, about 68% of the participants were between 14 and 16 years, matching the PISA instrument for 15-year-old students' everyday problem-solving (OECD, 2016).

Experimental and Control Groups

The experimental and control groups were distinguished by the infusion and withdrawal of the DECA, respectively. For the experimental group, the instructor promoted the application of DECA in the learning instructions and explicitly encouraged students to apply them in solving the programming tasks. For the control group, DECA was withdrawn and never highlighted in the instructions. All other conditions including the learning activities, direct instructions, scaffolding, and learning material were the same for both the experimental and control groups. For clarity, DECA is a localized strategy for solving problems and not a standalone pedagogical approach and can be infused into other instructional strategies.

Table 2 shows how DECA was implemented in the experimental group through worked example designed with a turtle (a module for drawing in Python) in drawing a cube. With the overarching aim of simplifying problem-solving, *discover* shows the underlying pattern as a component of two squares, four slanted lines and one circle. *Extract* examines the patterns for both relevant and irrelevant features. In this scenario, the circle was regarded as an irrelevant feature because the target is to draw a cube. *Create* enables the discovered patterns to be automated such as designing functions for drawing squares. *Assemble* represents how the patterns are combined systematically. By discovering patterns and extracting relevant features, learners

Table 1. Summary of the Profiles of the Participants.

| Participants Characteristics | Experimental Group <i>n</i> (%) | Control Group <i>n</i> (%) | Total <i>n</i> (%) |
|--|------------------------------------|-------------------------------|-----------------------|
| Enrolment | 31 | 28 | 59 |
| Gender | | | |
| Female | 7 (23) | 11 (39) | 18 (31) |
| Male | 24 (77) | 17 (61) | 41 (69) |
| Age | | | |
| Less than 14 years | 3 (10) | 4 (14) | 7 (12) |
| 14 – 16 years | 23 (74) | 17 (61) | 40 (68) |
| Greater than 16 years | 5 (16) | 7 (25) | 12 (20) |
| Computer-use proficiency | | | |
| Never used | 1 (3) | | 1 (2) |
| Novice | 11 (35) | 8 (29) | 19 (32) |
| Intermediate | 11 (35) | 15 (54) | 26 (44) |
| Advanced | 5 (16) | 4 (14) | 9 (15) |
| Expert | 3 (10) | 1 (4) | 4 (7) |
| Access to computers ^a | | | |
| Personal | 9 (29) | 12 (43) | 21 (36) |
| Parent | 8 (26) | 11 (39) | 19 (32) |
| Sibling | | 5 (18) | 5 (8) |
| School | 18 (58) | 13 (46) | 31 (53) |
| Prior programming experience | | | |
| Yes | 7 (23) | 10 (36) | 17 (29) |
| No | 24 (77) | 18 (64) | 42 (71) |
| Prior programming duration | | | |
| None | 24 (77) | 17 (61) | 41 (69) |
| Less than 1 year | 4 (13) | 11 (39) | 15 (25) |
| 1 – 2 years | 3 (10) | | 3 (5) |
| Prior programming environment ^a | | | |
| None | 26 (84) | 18 (64) | 44 (75) |
| Arduino | 3 (10) | 6 (21) | 9 (15) |
| Scratch | 2 (6) | 2 (7) | 4 (7) |
| Python | 1 (3) | | 1 (2) |
| Java | 1 (3) | | 1 (2) |
| C++ | | 1 (4) | 1 (2) |
| JavaScript | | 1 (4) | 1 (2) |
| Prior programming learning method ^a | | | |
| Self-learning | 4 (13) | 4 (14) | 8 (14) |
| School | 4 (13) | 5 (18) | 9 (15) |
| Tutoring centre | | 3 (11) | 3 (5) |
| Home | | 2 (7) | 2 (3) |

(continued)

Table 1. (continued)

| Participants Characteristics | Experimental Group <i>n</i> (%) | Control Group <i>n</i> (%) | Total <i>n</i> (%) |
|--|---------------------------------|----------------------------|--------------------|
| None | 24 (77) | 15 (54) | 39 (66) |
| Completed intervention and assessments | 20 (65) | 25 (89) | 45 (76) |

The percentage (%) is rounded to whole number and the total may not round to 100.

^aThe values for the participants' characteristics are not mutually exclusive.

Table 2. Application of Discover, Extract, Create, and Assemble (DECA) in a Worked Example.


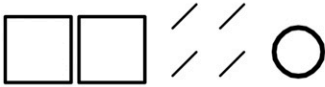


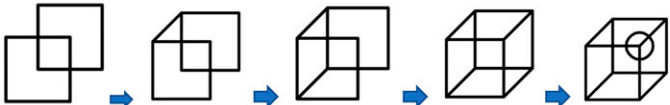

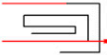



| Features | Description |
|----------|---|
| Project | Worked example Output – a cube drawn with turtle module in Python |
| |  |
| Discover | Examines entities or the operation of a process to uncover underlying patterns |
| |  |
| Extract | Focuses on distinguishing between the irrelevant, redundant, and relevant features |
| |  |
| Create | Synthesizes generic classification rules based on the pattern for automation. |
| |  |
| Assemble | Combines the relevant feature and common pattern systematically to form a functional unit |
| |  |

Table 3. Summary of Students' Target Programming Projects (Ezeamuzie, 2022).

| Project | Description |
|---|---|
|  Draw Polygon | Using the Turtle library, write a program to draw any regular polygon of your choice except triangle and square. For the girls' class, enhance the base program to accept the number of sides as input from the users. |
|  Moving Turtle | Using the Turtle library, write a program that initiates and controls the movement of a virtual turtle in different directions on the canvas. When users press the assigned keys, the turtle responds to the appropriate direction, start drawing, stop drawing, or change the colour. |
|  Draw Cube | Using the Turtle library, write a computer program to draw a three-dimensional object (e.g., cube) on the canvas. |
| <small>Python 3.6.1 on win32</small> Grade Point Calculator | Write a program that requests a student's final scores in three to eight subjects. It should calculate and display the student's total score, average score, and grade (based on pre-set criteria, such as 'if the average is greater than 80, assign the grade "A"'). |
| <small>Python 3.6.1 on win32</small> Point of Sale Simulator | Write a program to simulate the design of a point-of-sale system for a company. The company sells three products. For each day, the operator will set prices and the application will request quantities from customers' orders. Then, calculate and display the invoice for the orders in a table. |
|  Polygons in Quadrants | Using the Turtle library, write a program to divide the canvas into four quadrants. In each quadrant, draw any polygon with its centre in the middle of the quadrant. |
| <small>Python 3.6.1 on win32</small> Guessing Game | Write a program that asks users to guess a number between 1 and 100. Depending on the computer's chosen random number, users should be prompted to select a lower or higher number in every iteration. When the guess is correct, the program should display the number of attempts. |
|  Concentric Square | Using the Turtle library, write a program to draw two concentric squares on the canvas. |

focused on how to write codes for drawing a square and lines instead of a composite shape. The assembling activity drew from learners' knowledge of geometry for finding the length and angle of inclination of the lines and mapping the coordinates of the square.

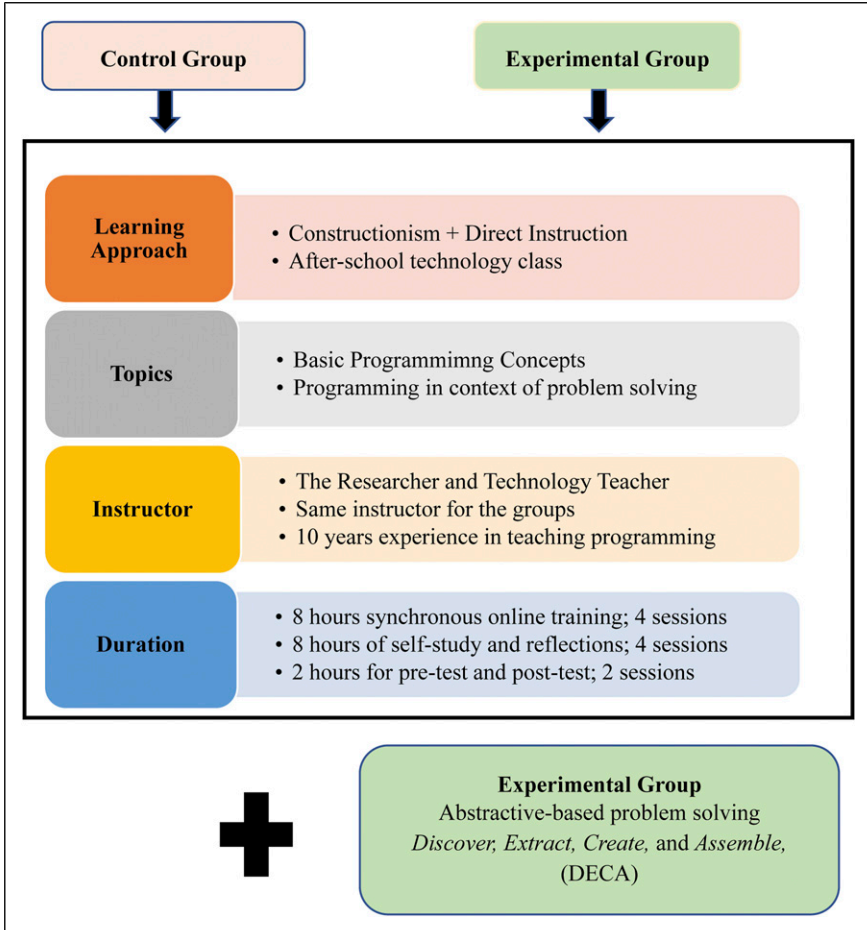


Figure 3. The research design and procedure.

Learning Activities and Resources

Constructionism was the preferred setting for all participants; challenging students to learn by doing. Although constructionism was the driving theory, all participants received direct instructions and scaffolding to support their learning. Appropriate learning goals were established for every lesson in the form of programming projects (see Table 3). By working towards the targets, students explored the intertwined relationships between programming concepts and different paths to solving problems. At the beginning of each class, the instructor simulated the day’s programming project without revealing the source code. This helped students understand the problems they

Table 4. Distribution of Selected Items by PISA 2012 Analytical and Assessment Framework.

| Dimension | Count | Credit | Question |
|----------------------|-------|--------|--------------------------|
| Process | | | |
| Employ | 3 | 6 | 13, 18,21 |
| Formulate | 3 | 6 | 6, 19, 20 |
| Interpret | 5 | 10 | 1, 2, 7, 11, 12 |
| Item type | | | |
| Constructed response | 7 | 14 | 1, 7, 11, 12, 19, 20, 21 |
| Multiple choice | 4 | 8 | 2, 6, 13, 18 |

Question 1: PENGUINS PM921Q01

Normally, a penguin couple produces two eggs every year. Usually the chick from the larger of the two eggs is the only one that survives.

With rockhopper penguins, the first egg weighs approximately 78 g and the second egg weighs approximately 110 g.

By approximately how many percent is the second egg heavier than the first egg?

A. 29%
 B. 32%
 C. 41%
 D. 71%




Figure 4. Penguin: sample question from PISA 2012.

would have to solve at the end of the class. Then, the instructor presented 1 – 2 mini-tasks that used similar concepts as worked examples. Students wrote codes to create different shapes and solve problems as assigned. Considering students' academic level, their prior exposure to programming, and the technical limitations of the settings, Python was selected as the programming language for the learning intervention. Turtle, a standard educational module in Python that is modelled after the Logo programming language for introductory programming, was used in some of the sessions.

Research Procedure

The learning intervention comprised of 2-hour weekly lessons that lasted for 10 weeks in the first term of the academic year. Randomization of groups was implemented through online enrolment for the after-school programming course. Students selected a preferred lesson day of either Tuesday or Thursday. Once the quota for any class day

The *Electrix Company* makes two types of electronic equipment: video and audio players. At the end of the daily production, the players are tested and those with faults are removed and sent for repair.

The following table shows the average number of players of each type that are made per day, and the average percentage of faulty players per day.

| Player type | Average number of players made per day | Average percentage of faulty players per day |
|---------------|--|--|
| Video players | 2000 | 5% |
| Audio players | 6000 | 3% |

Question 1: FAULTY PLAYERS PM00EQ01

Below are three statements about the daily production at *Electrix Company*. Are the statements correct?

Circle "Yes" or "No" for each statement.

| Statement | Is the statement correct? |
|---|---------------------------|
| One third of the players produced daily are video players. | Yes / No |
| In each batch of 100 video players made, exactly 5 will be faulty. | Yes / No |
| If an audio player is chosen at random from the daily production for testing, the probability that it will need to be repaired is 0.03. | Yes / No |

Figure 5. Faulty players: sample question from PISA 2012.

was reached, new registrants were asked to join another class if quota permits. The quota for each group was limited to 31 students; the number of internet-enabled computers in the computer lab. To avoid placebo effects, the nature of training or manipulation was concealed from participants.

A Virtual flipped learning setting was adopted for the after-school training. In the first week, the instructor explained the aims of the lessons including the schedule, duration and rights to join or decline participation. After the brief introduction, students took part in an hour-long online pre-test. Participants attended four online lessons on the second, fourth, sixth, and eighth week respectively via Zoom (a group video-conferencing application; <https://zoom.us/>). For the third, fifth, seventh and ninth weeks, programming tasks were assigned to the participants for self-learning and

Table 5. Distribution of Items in the Programming Knowledge Assessment.

| Programming Type | Count | Unscaled Credit | Question |
|----------------------|-------|-----------------|--------------|
| Code writing | 3 | 12 | 5, 10, 16 |
| Code reading | | | |
| Constructed response | 4 | 8 | 4, 9, 15, 17 |
| Multiple choice | 3 | 3 | 3, 8, 14 |

practice. The 10th week was dedicated to an hour-long online post-test. Figure 3 shows the summary of the research design and procedure.

Data Collection

Three instruments – demographic questionnaire, CT assessment and programming knowledge assessment constituted the primary data collection instruments. These instruments were administered at various stages of the experiment using Qualtrics (an online survey solution; <https://qualtrics.com>). Each participant was assigned a unique ID during registration, which was used to track and collate their pretest and posttest results for analysis.

Demographic questionnaire. A simple questionnaire was administered to capture basic demographic data (e.g., age, gender), computer proficiency and prior programming experience of the participants. Participants completed the questionnaire before the commencement of the study (see [Supplementary 1](#), available from the author and in the Online version)

Computational thinking assessment. PISA drew from the massive experience of testing centres and a large pool of experts to design test items focused on problem-solving within the mathematical literacy framework that reflects on real-life challenges and situations. Items in PISA were selected from pools that were piloted in all the participating countries (OECD, 2014). To build the pool of items, participating national centres submitted sample test items, totalling about 500 items, which were subjected to rigorous checks including national item review, international item review, refining by testing centres and reviewing by Mathematics and Problem-Solving Expert Groups. Items were selected if they have good psychometric properties, had no coding issues in field trials, were rated high by national centres, aligned to the domain, and item difficulties were well distributed. Based on the item response theory, PISA has excellent reliabilities as follows: overall ($\alpha = .914$), employing ($\alpha = .909$), formulating ($\alpha = .892$), and interpreting ($\alpha = .897$). Amongst the 110 mathematics items in PISA 2012 final pool, 26 items were released (OECD, 2014) and a subset of 11 items were adopted for the pre-test and post-test. Table 4 shows the distribution of the CT items in the pre-test and post-test. The full test is available as [Supplementary 2](#) (available from the author

Table 6. Interrater Reliability Matrix Using the Intraclass Correlation Coefficient.

| Dimension | Pretest | Posttest |
|-------------------------------------|---------|----------|
| Computational Thinking | .991 | .996 |
| Employ | .990 | 1.000 |
| Formulate | 1.000 | 1.000 |
| Interpret | .974 | .985 |
| Programming | .993 | .987 |
| Code writing | .983 | .968 |
| Code reading (constructed response) | .990 | .983 |
| Code reading (multiple choice) | 1.000 | 1.000 |

Table 7. Data Analysis by the Research Questions, the Statistical Test, Stage of Data Collection and Participants Groups.

| Research Question | Statistical Test | Stage of Data Collection | Participants' Group |
|---|--------------------|--------------------------|-----------------------|
| Equality of groups at baseline | Independent t-test | Pretest | Control, experimental |
| RQ 1 - the effect of abstractive-based programming on CT | Independent t-test | Posttest | Control, experimental |
| | Paired t-test | Pretest, posttest | Control |
| | Paired t-test | Pretest, posttest | Experimental |
| RQ 2 - the effect of abstractive-based programming on programming knowledge | Independent t-test | Posttest | Control, experimental |
| | Paired t-test | Pretest, posttest | Control |
| | Paired t-test | Pretest, posttest | Experimental |
| RQ 3 - the interaction effects of abstractive-based programming and students' characteristics | Two-way ANOVA | Posttest | Control, experimental |

and in the Online version). Following PISA's rubric for grading, maximum score of 2-points was assigned for correct answers, 1-point for a partially correct answer and none for other responses. Figures 4 and 5 are sample questions from PISA 2012 on penguin reproduction habits and faulty video and audio players, respectively.

The penguin reproduction habit (Figure 4) requires students to compare and convert the weights of two eggs, which fits into the quantity content dimension. In context, it deals with the issue of population growth and biodiversity and is classified as scientific. On the process dimension, it fits the employing practice, which requires applying

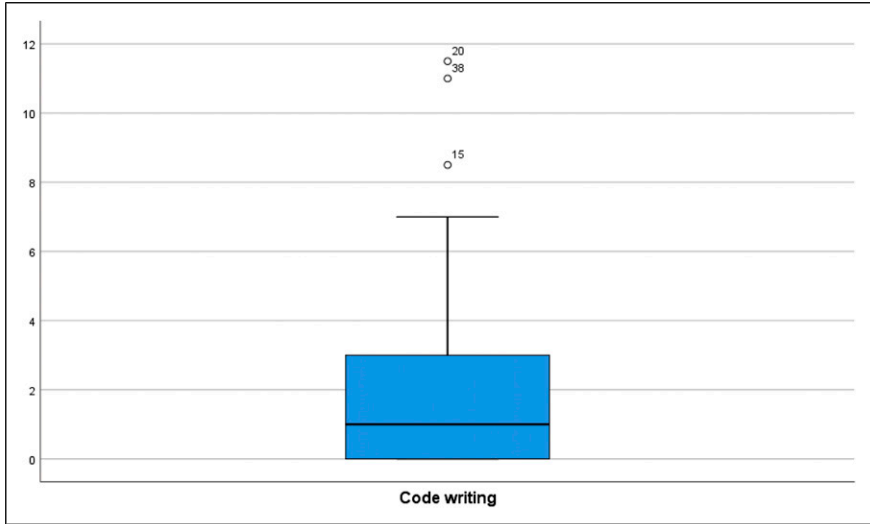


Figure 6. Boxplot of pretest code writing depicting three outliers in the control group.

mathematical formulas to calculate the percentage. The correct answer of 41% (option C) received the 2 maximum points. Other responses were graded 0.

The faulty video and audio players' question (Figure 5) is a problem one will expect in the manufacturing industry. This question requires students to comprehend statistical data and interpret probability statements. Since it requires manipulating uncertainty, it maps to the uncertainty and data in the content dimension. Context is occupational as it deals with issues facing manufacturers. On the process dimension, it fits the formulating practice because it requires creating mathematical models to validate or refute the claims. The correct answer in the order of 'No, No, Yes' received the 2 maximum points. Other responses were graded 0.

Programming knowledge assessment

Programming knowledge was classified into *procedural* or *strategic* knowledge (Lau & Yuen, 2009). Procedural knowledge (code reading) deals with comprehension of syntax/semantics and interpretation of program logic. Items in the procedural category required either selecting from multiple choices or constructed responses. Strategic knowledge (code writing) entails putting the pieces of program concepts together to achieve a specific goal through writing codes. Each item posed a problem scenario and tasked participants to develop a written program solution. Three questions that require writing a Python program were selected for this study from a pool of 15 programming tasks that were piloted with students of similar age grades (Ezeamuzie, 2022). Items were designed to assess participants' programming knowledge within the content area covered in the learning intervention.

Table 8. Posttest of Computational Thinking: Mean, Standard Deviation and Mean Difference.

| | Experimental | | | Control | | | Mean Diff | t | p |
|------------------------|--------------|--------|--------|---------|--------|--------|-----------|--------|------|
| | N | M | SD | N | M | SD | | | |
| Computational thinking | 20 | 10.450 | 5.8869 | 22 | 10.364 | 4.1581 | .0864 | .055 | .956 |
| Employ | 20 | 3.50 | 2.14 | 22 | 2.55 | 1.654 | .955 | 1.626 | .112 |
| Formulate | 20 | 1.90 | 1.651 | 22 | 1.55 | 1.625 | .355 | .701 | .487 |
| Interpret | 20 | 5.050 | 2.8695 | 22 | 6.273 | 2.2078 | -1.2227 | -1.556 | .128 |

Each of the selected tasks requires between 4 – 10 minutes for students to write a complete programmatic solution. Isomorphic representations of the three programming tasks were created to probe whether problem-solving through code reading and code writing have different influences on performance. Also, understand the impact of multiple-choice solutions and constructed responses on procedural programming knowledge. With the isomorphic representation of the three questions, 10-item programming tasks were adopted for measuring participants' programming knowledge. Table 5 shows the distribution of the programming task items in the pre-test and post-test. The test items are available as Supplementary 2. The multiple-choice option received one point for correct answers and zero for wrong responses. For the code reading tasks with a constructed response, two points were awarded for correct answer, one point for a partially correct answer and zero for the incorrect responses. The code writing tasks were assigned a maximum of four points based on the following criteria – (a) two points for the correct logic (b) one point for correct output or displaying implementation, and (c) one point for correct syntax and application of variables.

Data Analysis

The researcher and the technology teacher graded the students programming and CT responses independently. The intraclass correlation coefficient (Shrout & Fleiss, 1979), an effective approach for measuring interrater reliability for continuous data was computed. Table 6 shows the intraclass correlation coefficients of CT ability and programming knowledge. With the high consistency between the researcher's and technology teachers' ratings ($p < .001$), the effective scores were computed as the average of the graded score.

Table 7 shows the mapping of data analysis by the research questions, the statistical test, stage of data collection and participant groups.

Results

Outliers, Homogeneity of Variance and Normal Distribution of Data

To use ANOVA and t-test, the data were tested for parametric conditions of equality of population variance across the various groups and normally distributed data in each

Table 9. Posttest of Programming Knowledge: Mean, Standard Deviation and Mean Difference.

| | Experimental | | | Control | | | Mean Diff | T | p |
|--|--------------|-------|--------|---------|--------|--------|-----------|--------|------|
| | N | M | SD | N | M | SD | | | |
| Programming | 20 | 9.875 | 7.5601 | 22 | 10.818 | 7.4858 | -.9432 | -.406 | .687 |
| Code writing | 20 | 4.125 | 3.9930 | 22 | 4.250 | 4.1483 | -.1250 | -.099 | .921 |
| Code reading (constructed response) | 20 | 4.100 | 2.9540 | 22 | 4.523 | 2.9052 | -.4227 | -.467 | .643 |
| Code reading (multiple choice) | 20 | 1.65 | 1.137 | 22 | 2.05 | 1.174 | -.395 | -1.107 | .275 |

Table 10. Paired Samples Test and Mean Difference Between Posttest and Pretest Scores.

| | Experimental | | | Control | | |
|-------------------------------------|--------------|--------|------|-----------|--------|------|
| | Mean Diff | SD | P | Mean Diff | SD | p |
| Computational Thinking | 1.5500 | 3.2803 | .048 | 1.4318 | 4.5179 | .152 |
| Employ | 1.200 | 1.642 | .004 | -.045 | 2.149 | .922 |
| Formulate | .100 | 1.373 | .748 | .364 | 1.590 | .296 |
| Interpret | .2500 | 1.5087 | .468 | 1.1136 | 2.6679 | .064 |
| Programming | 4.2250 | 6.3339 | .008 | 6.2273 | 6.8378 | .000 |
| Code writing | 2.7500 | 2.8074 | .000 | 2.6818 | 3.4280 | .001 |
| Code reading (constructed response) | 1.3250 | 2.9122 | .056 | 2.7727 | 3.0812 | .000 |
| Code reading (multiple choice) | .150 | 1.309 | .614 | .773 | 1.412 | .018 |

group of the independent variables. Also, the dependent variables were on a continuous scale and each case was observed independently. Data were also observed for outliers in the following dependent variables. Signs of outliers emerged from the pretest programming knowledge and were further investigated. Figure 6 shows three outliers on pretest code writing ($M = 2.067$, $SD = 3.0125$). The outlier data points (15, 38, and 20) have scores of code writing values of 8.5, 11, and 11.5, respectively depicting that the three participants' code writing abilities were not at the same baseline as the group. They were excluded from data analysis.

Equality of Groups at Pretest: Computational Thinking and Programming Knowledge

Establishing whether the control and experimental groups are equal at the baseline is a prerequisite for experimental study. Using an independent t-test with two levels of learning group factor (experimental and control), the pretest score of the participants' CT and programming knowledge were analysed. At the $p < 0.05$ level, no significant difference was observed between the control and experimental groups in the overall CT

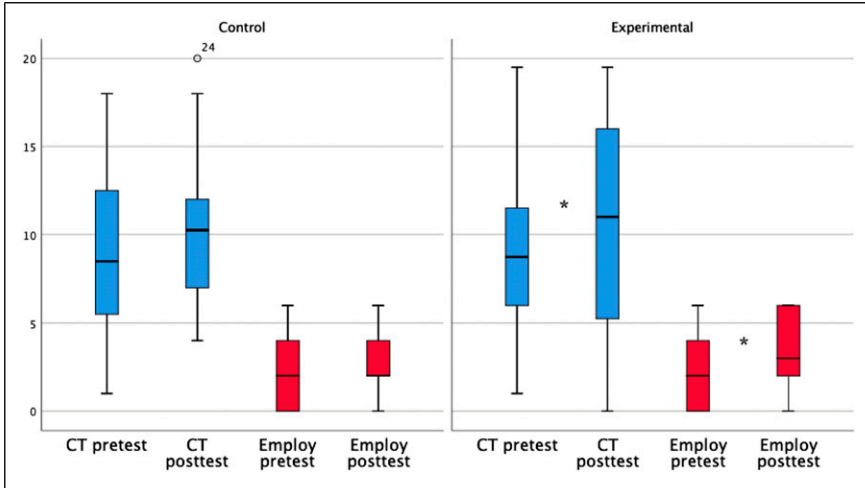


Figure 7. Paired sample test for overall computational thinking and employing mathematical knowledge in problem solving writing. CT = Computational thinking. * $p < .05$.

($t(40) = .024, p = .981$), employ ($t(40) = .460, p = .648$), formulate ($t(40) = -1.262, p = .214$), and interpret ($t(40) = .491, p = .626$). Also, no significant difference was observed between the control and experimental groups in the overall programming knowledge ($t(40) = -.730, p = .470$), code writing ($t(40) = .300, p = .766$), code reading in constructed response ($t(40) = -1.421, p = .163$), and code reading in multiple-choice questions ($t(40) = -.723, p = .474$). Therefore, the control and experimental groups were deemed to have similar CT ability and programming knowledge at baseline.

Posttest of Computational Thinking

An independent t-test with two levels of learning group (experimental and control) was conducted to analyse the CT score. Table 8 shows the posttest result of CT ability. At $p < 0.05$ level, no significant difference was observed between the control and experimental groups in the employ, formulate, interpret and overall CT ability.

Posttest of Programming Knowledge

An independent t-test with two levels of learning group (experimental and control) was conducted to analyse the programming knowledge score. Table 9 shows the posttest result of programming knowledge. At $p < 0.05$ level, no significant difference was observed between the control and experimental groups in code writing, code reading and overall programming knowledge.

Table 11. Age and Learning Group Interaction: 2-way ANOVA of Computational Thinking.

| Source | Type III Sum of Squares | Df | Mean Square | F | p |
|----------------------|-------------------------|----|-------------|-------|------|
| Age | 25.180 | 2 | 12.590 | .564 | .574 |
| Learning group | .249 | 1 | .249 | .011 | .916 |
| Age * learning group | 194.310 | 2 | 97.155 | 4.352 | .021 |
| Error | 758.997 | 34 | 22.323 | | |
| Total | 5370.250 | 40 | | | |
| Corrected Total | 970.744 | 39 | | | |

R Squared = .218 (Adjusted R Squared = .103).

Table 12. Computer Proficiency and Learning Group Interaction: 2-way ANOVA of Computational Thinking.

| Source | Type III Sum of Squares | df | Mean Square | F | p |
|---------------------------------------|-------------------------|----|-------------|--------|-------|
| Computer proficiency | 258.338 | 1 | 258.338 | 13.660 | <.001 |
| Learning group | 1.838 | 1 | 1.838 | .097 | .757 |
| Computer proficiency * learning group | 31.538 | 1 | 31.538 | 1.668 | .205 |
| Error | 680.813 | 36 | 18.911 | | |
| Total | 5370.250 | 40 | | | |
| Corrected Total | 970.744 | 39 | | | |

R Squared = .299 (Adjusted R Squared = .240).

Note: Independent t-test analysis shows that significant difference exists between non-beginners and beginners (Mean Difference = 5.188, SE=1.404, $p < .001$).

Within Group Change (Posttest vs. Pretest Comparison): Computational Thinking and Programming Knowledge

A further test was conducted to understand the effect of the learning intervention within the groups (i.e., from pretest to posttest). At $p < .05$ level, [Table 10](#) shows the paired sample t-test result and the mean difference in CT and programming knowledge. The mean scores improved in all dimensions except the control group's ability to employ mathematical knowledge in solving problems, which decline from the pretest.

The mean difference in programming knowledge was significant in both the control and experimental group. Also, both groups showed significant improvement in code writing ability after the intervention. Unlike the control group, the difference in code reading was not significant in the experimental group.

On CT performance, no significant difference was observed between the posttest and pretest CT ability of the control groups. Conversely, the development in the experimental group's CT ability was significant in employing mathematical knowledge in problem-solving and the overall CT, accounting for about 20% and 7% improvement

Table 13. Computer Proficiency and Learning Group Interaction: 2-way ANOVA of Programming.

| Source | Type III Sum of Squares | df | Mean Square | F | p |
|---------------------------------------|-------------------------|----|-------------|-------|------|
| Computer Proficiency | 403.004 | 1 | 403.004 | 8.460 | .006 |
| Learning group | 20.417 | 1 | 20.417 | .429 | .517 |
| Computer proficiency * learning group | 8.067 | 1 | 8.067 | .169 | .683 |
| Error | 1714.917 | 36 | 47.637 | | |
| Total | 6562.750 | 40 | | | |
| Corrected Total | 2142.244 | 39 | | | |

R Squared = .199 (Adjusted R Squared = .133).

Note: Independent t-test analysis shows that significant difference exists between non-beginners and beginners (Mean Difference = 6.479, SE=2.228, $p = .006$).

on the assessment scale, respectively. [Figure 7](#) compares the improvement in CT and employing mathematical knowledge from the pretest and posttest of both the experimental and control groups.

Interaction Between Learning Approach and Participants' Characteristics

Since the CT assessment was adopted from the PISA instrument for 15-year-old students, further analysis was conducted to check for any interaction effect between learning group, age, and gender. Also, the composition of the participant from developing regions prompted an investigation on whether prior computer usage or programming experience influenced the development of CT and programming knowledge.

Age and learning group interaction

Using two-way ANOVA with two levels of learning group (experimental and control) and three levels of age (less than 14 years, 15 years, and greater than 16 years), CT and programming knowledge were analysed. The 15-year group identified their age as between 14 and 15 years or between 15 and 16 years.

[Table 11](#) is the two-way ANOVA result of the effect of age and learning group on CT at the $p < 0.05$ level. No main effect of age was found ($F(2, 40) = .564, MS=12.590, p = .574$). However, there was a statistically significant interaction between the effects of age and learning group on CT ($F(2, 40) = 4.352, p = .021$). To identify where differences exist, a pairwise comparison of the control and experimental groups within each age level shows that the 16+ years old in the control group had significant difference development in CT than their experimental group counterpart (Mean Difference = 7.833, SE=3.450, $p = .030$).

Two-way ANOVA result of the effect of age and learning group on programming at the $p < 0.05$ level shows no main effect of age ($F(2, 40) = .783, MS=38.551, p = .465$) and no significant interaction effects on programming ($F(2, 40) = 3.234, p = .052$).

Computer proficiency and learning group interaction. Using two-way ANOVA with two levels of learning group (experimental and control) and two levels of computer proficiency (beginner and non-beginner), CT and programming knowledge were analysed. The beginner comprised of participants that identified their computer proficiency as 'never used' or 'novice user'. The non-beginner represents participants that have intermediate, advanced, or expert exposure to a computer.

Table 12 is the two-way ANOVA result of the effect of computer proficiency and learning group on CT at the $p < 0.05$ level. There is a main effect of computer proficiency ($F(1, 40) = 13.660, MS=258.338, p < .001$) on CT. To identify the difference in computer proficiency on CT, an independent t-test analysis to compare the non-beginners and beginners was performed. A significant difference was found between non-beginners and beginners (Mean Difference = 5.188, $SE=1.404, p < .001$). There was no statistically significant interaction between the computer proficiency and learning group on CT ($F(1, 40) = 1.668, p = .205$).

Table 13 is the two-way ANOVA result of the effect of computer proficiency and learning group on programming at the $p < 0.05$ level. There is a main effect of computer proficiency ($F(1, 40) = 8.460, MS=403.004, p = .006$) on programming knowledge. To identify the difference in computer proficiency on programming knowledge, an independent t-test analysis to compare the non-beginners and beginners was performed. A significant difference was found between non-beginners and beginners (Mean Difference = 6.479, $SE=2.228, p = .006$). There was no interaction between the computer use proficiency and learning group on programming ($F(1, 40) = .169, p = .683$).

Prior programming experience and learning group interaction. Using two-way ANOVA with two levels of learning group (experimental and control) and two levels of programming experience (beginner and non-beginner), CT and programming knowledge were analysed. Two-way ANOVA result of the effect of prior programming experience and learning group on CT at the $p < 0.05$ level shows no main effect of prior programming experience ($F(1, 40) = 2.829, MS = 69.952, p = .101$) and no significant interaction between the prior programming experience and learning group on CT ($F(1, 40) = .668, p = .419$). Also, two-way ANOVA result of the effect of prior programming experience and learning group on programming at the $p < 0.05$ level shows no main effect of prior programming experience ($F(1, 40) = .549, MS= 31.515, p = .463$) and no significant interaction between the prior programming experience and learning group on programming ($F(1, 40) = .394, p = .534$).

Gender and learning group interaction. Using two-way ANOVA with two levels of learning group (experimental and control) and two levels of gender (female and male),

CT and programming knowledge were analysed. Two-way ANOVA result of the effect of gender and learning group on CT at the $p < 0.05$ level shows no main effect of gender ($F(1, 40) = 2.419$, $MS = 61.112$, $p = .129$) and no significant interaction between the gender and learning group on CT ($F(1, 40) = .011$, $p = .918$). Also, two-way ANOVA result of the effect of gender and learning group on programming at the $p < 0.05$ level shows no main effect of gender ($F(1, 40) = .347$, $MS = 20.000$, $p = .560$) and no significant interaction between the gender and learning group on programming ($F(1, 40) = .434$, $p = .514$).

Discussion

The central issue of investigation in this study is whether learning to program using DECA abstractive-based instructional model influence students' CT skills (RQ1) and programming knowledge (RQ2). Also, supplementary research question probed for interaction effects of students' characteristics (age, computer proficiency, prior programming experience, gender) towards CT (RQ3). This section summarises the results, examines how the findings support or contradict prior studies, and highlights the implication of the findings.

Impact of Abstractive-Based Learning on Computational Thinking (RQ 1) and Programming Knowledge (RQ 2)

In this study, the evidence that learning programming using the DECA model supports students' programming and CT skills is mixed and needs to be interpreted with caution. According to [Table 8](#), the posttest comparison of CT shows no significant difference between the control and experimental groups in both the overall CT and the sub-dimensions of employ, formulate, and interpret. Similarly, in the development of programming knowledge as depicted in [Table 9](#), no significant difference was observed between the control and experimental groups in code writing, code reading and overall programming knowledge.

Although there was no significant difference between the experimental and control groups in CT skill and programming knowledge at posttest, further investigation through paired sample t-test (see [Table 10](#)) showed that students improved in all dimensions except the decline in the 'employ' dimension of CT in the control group. Concerning programming knowledge, the mean difference between overall programming knowledge in posttest and pretest was significant in both the control and experimental group. For CT skills, only the experimental group had significant improvement, which implies that DECA was effective in developing CT. This finding is interpreted with caution because no significant difference was observed in the between-groups independent t-test comparison at posttest. Although DECA resulted in significant development of CT in the within-group analysis, the result is more appropriately described as inconclusive evidence because there was no significant

difference between the control and experimental groups at posttest, after equality of groups was confirmed at pretest.

On the arguments about learning programming and the transfer of problem solving in other domains, [Pea and Kurland \(1984\)](#) cast doubt on the perception of the cognitive benefits of programming. However, a meta-analysis showed that learning programming improves students' cognitive ability ([Liao, 2000](#); [Liao & Bright, 1991](#)). Claims of transfer from programming have been refuted in recent years ([Denning, 2017](#); [Guzdial, 2015](#)). But [Scherer et al.'s \(2019\)](#) meta-analytic finding suggested that the contentious issue of transfer of programming to CT is substantiated, which contrasts with the findings in this study.

Decoupling CT from programming is an important element in this study that reflects [Wing's \(2006\)](#) position on CT as 'conceptualizing, not programming' (p. 35) but contrasts with [Scherer et al.'s \(2019\)](#) classification of CT as a near transfer skill (i.e., part of the programming knowledge). Therefore, probing what was measured is a critical argument about the transferability of programming. For instance, [Wei et al. \(2021\)](#) examined the effectiveness of partial pair programming (PPP) on elementary school students' CT skills with Dr Scratch ([Moreno-León et al., 2015](#)) and [Taylor and Baek \(2019\)](#) investigated the impact of grouping by gender and group roles on CT skills with the CT-test by [Román-González et al. \(2017\)](#). Although [Wei et al. \(2021\)](#) and [Taylor and Baek \(2019\)](#) found positive transfer outcomes, code writing and code reading were the underlying skills measured in the studies. Other studies adopted self-report instruments such as the CT-scale by [Korkmaz et al. \(2017\)](#) in an investigation of the effects of critical reflection on middle schoolers' CT skills ([He et al., 2021](#)). Whereas [He et al. \(2021\)](#) found that students' CT skills improved, the finding seems to equate CT disposition and self-beliefs to represent CT skills. When CT is operationalized as programming or disposition, it may not be comparable with the result of CT as an instance of everyday problem-solving.

In studies that adopted CT assessments that mirrored real-life contexts, Bebras tasks ([Dagienė & Sentance, 2016](#)) were adopted to investigate the effect of robotic programming on primary schoolers' CT skills ([Noh & Lee, 2020](#)). In a similar investigation but on the influence of virtual robotic programming in middle school, CT assessment was modelled as a real-life scenario of applying sensors in tracking vehicular capacity on a bridge ([Witherspoon et al., 2017](#)). A study by [Akcaoglu \(2014\)](#) adopted PISA in assessing students' problem-solving after a vacation game-making program. These studies ([Akcaoglu, 2014](#); [Noh & Lee, 2020](#); [Witherspoon et al., 2017](#)) modelled CT in real-life contexts and reported significant improvements in students' CT skills. The findings of the above studies contradict the inconclusive transfer effect of programming on CT in this study. However, closer analysis revealed that the studies adopted one group pretest-posttest research design. Therefore, the significant CT improvements in the studies is identical and consistent with the paired t-test analysis (see [Table 10](#)) that showed significant CT improvement among the experimental group but not the control group. Implicitly, using abstractive-based programming – DECA resulted in significant improvement when posttest is compared with pretest.

Nonetheless, the issue of transfer is considered inconclusive as the experimental standard of comparing between groups (i.e., control vs. experimental at the posttest, with homogeneity at baseline) showed that the difference is not significant.

With respect to programming knowledge, the control and experimental groups recorded significant improvement between the pretest and posttest after 8 hours of self-study and another 8 hours of synchronous online training. This improvement shows that problem-based learning, when supported with worked examples and appropriate scaffolding, constitutes an effective approach to learning programming. This is consistent with the finding that students' programming ability remained high from the first lesson of constructionist learning (Ezeamuzie, 2022) and the strong near-transfer ($g = .75$) between learning programming and programming knowledge (Scherer et al., 2019).

Interaction Effects of Abstractive-based Programming and Students' Characteristics on Computational Thinking and Programming Knowledge (RQ 3)

Analysis of the interaction effects of approach to learning programming and students' characteristics showed that prior programming experience and gender had neither significant interaction nor main effect on CT and programming knowledge. These findings are consistent with prior studies that found no effect of gender on programming (Grover et al., 2019; Lau & Yuen, 2009; 2011; Sullivan & Bers, 2019).

An interaction effect was found between the learning approach and age on CT, suggesting that students in the control group with ages greater than 16 years outperformed their experimental group equivalent. However, this is interpreted with caution because of the small number of participants that are older than 16 years in the experimental group ($n = 3$). Out of the three students in the experimental group with ages greater than 16 years, compared with five in the control group, further analysis of time spent on the posttest showed that one of the students in the experimental group completed less than 38% of the test. The finding of no main effect by age contrast with the cross-sectional study that examined the difference in CT between lower and upper elementary school students (Kyza et al., 2022). Nonetheless, the findings in Kyza et al. (2022) that upper elementary students outperformed their peers in lower classes with significant differences in abstraction and decomposition may not be comparable as the assessment was based on Dr Scratch's analysis of programming artefacts. Also, another plausible explanation is the wide developmental difference between the participants in Kyza et al. (2022) and the near uniformity in the age of participants in this study.

A noticeable exception is the main effect of computer proficiency, implying that mastery in the use of computers has a positive impact on the development of CT and programming knowledge. To make sense of this result, beginners comprised participants who self-reported their computer-use proficiency as 'never used' or 'novice user'. The non-beginner identified their exposure to computers as intermediate, advanced, or expert. In addition (see Table 1), this investigation was implemented in a

region where a sizeable number of the participants have minimal or no exposure to technology-enhanced learning. Within this context, other plausible interpretations exist. For example, the difference may not be truly a reflection of programming knowledge. The non-beginners familiarity with computers may be the edge in performance as learning and assessment were conducted through computers.

Logically, one may expect prior programming experience to influence CT and programming knowledge. While this was never the case, the composition of participants and self-reported prior programming experience provide some clues. [Table 1](#) shows that 29% of participants reported prior programming experience. However, about 95% of participants have no programming experience or have programmed for less than 1 year. Considering participants' exposure to computers, it is plausible that their reported prior programming experience may be isolated exposure rather than a persistent learning experience.

Conclusion, Limitations, and Implications

As claims and counterclaims about the transfer effect of programming on cognitive skills have persisted across centuries, this study was designed to provide further clarity on the transferability of CT through learning programming. Specifically, operationalizing CT as problem-solving in the PISA framework that is well segregated from programming. More so, this study introduced DECA – a novel and systematic way of solving programming tasks that is grounded in abstraction that may enhance transfer to CT. Gaining a holistic understanding of the interaction of DECA and CT, when disentangled from programming, is a crucial step in advancing research and practice on both programming and CT.

To highlight the educational implications of this study, it is important to acknowledge its limitations. The major challenge was conducting this experiment during the COVID-19 pandemic. Some of the research plans were adjusted to meet the prevailing restrictions. For example, the original plan for in-class training was switched to synchronous online learning. In addition, plans to collect supplementary qualitative data through student interviews and in-class observation for triangulation were affected too. Social distancing regulations and school prevailing policy including the compulsory vacation of students after the training hindered the conduct of interviews. Bottlenecks in networks and online learning hindered the observation of students' programming processes, attitudes, and challenges in an authentic setting.

Despite these limitations, this study unearthed valuable lessons for both practice and research. Theoretically, the abstractive-based model – DECA was operationalized as a pedagogical approach to provide systematic processes for solving problems. By decontextualizing CT from programming in this study through PISA, it uncovered several trouble spots that hinder the generalization that learning programming enhances CT and expounded gaps for future studies to consider in operationalizing CT. Although evidence that learning programming enhances CT is inconclusive, the within-group improvement in the experimental group's CT and the lack of it in the control group are

signs that infusing DECA in problem-solving is a promising instructional approach. Researchers will have a deeper understanding of how CT is conceptualized and stimulate more empirical studies on developing CT skills in 21st century learners.

What was unearthed in this study revealed the potential of CT in supporting cross-disciplinary and everyday problem-solving; an educational goal that Jonassen (2000) described as ‘the most important outcome for life’ (p. 63). Nonetheless, more research is needed to articulate how computer scientists think and how they relate to everyday problems. Considering the limitations in this study such as duration of learning, limited technology and students’ unfamiliarity with synchronous online learning, more investigation is required to explore the role of abstraction such as DECA, which provides a systematic way of solving problems.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

ORCID iD

Ndudi O. Ezeamuzie  <https://orcid.org/0000-0001-8946-5709>

Supplemental Material

Supplemental material for this article is available online.

References

- Akcaoglu, M. (2014). Learning problem-solving through making games at the game design and learning summer program. *Educational Technology Research and Development*, 62(5), 583–600. <https://doi.org/10.1007/s11423-014-9347-4>
- Baek, Y., Wang, S., Yang, D., Ching, Y., Swanson, S., & Chittoori, B. (2019). Revisiting second graders’ robotics with an understand/use-modify-create (U²MC) strategy. *European Journal of STEM Education*, 4(1), Article 07. <https://doi.org/10.20897/ejsteme/5772>
- Barr, D., Harrison, J., & Conery, L. (2011). Computational thinking: A digital age skill for everyone. *Learning Leading with Technology*, 38(6), 20–23. https://www.learningandleading-digital.com/learning_leading/20110304?pm=2&pg=22#pg22
- 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. <https://doi.org/10.1145/1929887.1929905>

- 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(C), 145–157. <https://doi.org/10.1016/j.compedu.2013.10.020>
- Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*, 1, 1–25. <http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf>
- Chen, G., Shen, J., Barth-Cohen, L., Jiang, S., Huang, X., & Eltoukhy, M. (2017). Assessing elementary students' computational thinking in everyday reasoning and robotics programming. *Computers & Education*, 109, 162–175. <https://doi.org/10.1016/j.compedu.2017.03.001>
- Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines*, 17(2), 169–184. <https://doi.org/10.1007/s11023-007-9061-7>
- Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., & Selby, C. (2015). *Computational thinking: A guide for teachers*. <https://eprints.soton.ac.uk/424545/>
- Dagienė, V., & Sentance, S. (2016). It's computational thinking! Bebras tasks in the curriculum. In A. Brodnik & F. Tort (Eds.), *Informatics in schools: Improvement of informatics knowledge and perception* (pp. 28–39). Springer. https://doi.org/10.1007/978-3-319-46747-4_3
- Denner, J., Campe, S., & Werner, L. (2019). Does computer game design and programming benefit children? A meta-synthesis of research. *ACM Transactions on Computing Education*, 19(3), Article 19. <https://doi.org/10.1145/3277565>
- Denning, P. J. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6), 33–39. <https://doi.org/10.1145/2998438>
- Denning, P. J., Tedre, M., & Yongpradit, P. (2017). Misconceptions about computer science. *Communications of the ACM*, 60(3), 31–33. <https://doi.org/10.1145/3041047>
- Ezeamuzie, N. O. (2022). Project-first approach to programming in K–12: Tracking the development of novice programmers in technology-deprived environments. *Education and Information Technologies*. <https://doi.org/10.1007/s10639-022-11180-8>
- Ezeamuzie, N. O., & Leung, J. S. C. (2022). Computational thinking through an empirical lens: A systematic review of literature. *Journal of Educational Computing Research*, 60(2), 481–511. <https://doi.org/10.1177/07356331211033158>
- Ezeamuzie, N. O., Leung, J. S. C., Garcia, R., & Ting, F. S. T. (2022a). Discovering computational thinking in everyday problem solving: A multiple case study of route planning. *Journal of Computer Assisted Learning*. <https://doi.org/10.1111/jcal.12720>
- Ezeamuzie, N. O., Leung, J. S. C., & Ting, F. S. T. (2022b). Unleashing the potential of abstraction from cloud of computational thinking: A systematic review of literature. *Journal of Educational Computing Research*, 60(4), 877–905. <https://doi.org/10.1177/07356331211055379>
- Grover, S., Jackiw, N., & Lundh, P. (2019). Concepts before coding: Non-programming interactives to advance learning of introductory programming concepts in middle school. *Computer Science Education*, 29(2–3), 106–135. <https://doi.org/10.1080/08993408.2019.1568955>

- Guzdial, M. (2008). Paving the way for computational thinking. *Communications of the ACM*, 51(8), 25–27. <https://doi.org/10.1145/1378704.1378713>
- Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6), 1–165. <https://doi.org/10.2200/S00684ED1V01Y201511HCI033>
- He, Z. Z., Wu, X. M., Wang, Q. Y., & Huang, C. Q. (2021). Developing eighth-grade students' computational thinking with critical reflection. *Sustainability*, 13(20), Article 11192. <https://doi.org/10.3390/su132011192>
- Jonassen, D. H. (1997). Instructional design models for well-structured and III-structured problem-solving learning outcomes. *Educational Technology, Research and Development*, 45(1), 65–94. <https://doi.org/10.1007/BF02299613>
- Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4), 63–85. <https://doi.org/10.1007/bf02300500>
- Knuth, D. E. (1974). Computer science and its relation to mathematics. *The American Mathematical Monthly*, 81(4), 323–343. <https://doi.org/10.1080/00029890.1974.11993556>
- Korkmaz, Ö., Çakir, R., & Özden, M. Y. (2017). A validity and reliability study of the computational thinking scales (CTS). *Computers in Human Behavior*, 72, 558–569. <https://doi.org/10.1016/j.chb.2017.01.005>
- Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4), 36–42. <https://doi.org/10.1145/1232743.1232745>
- Kyza, E. A., Georgiou, Y., Agesilaou, A., & Souropetsis, M. (2022). A cross-sectional study investigating primary school children's coding practices and computational thinking using ScratchJr. *Journal of Educational Computing Research*, 60(1), 220–257. <https://doi.org/10.1177/07356331211027387>
- Lau, W. W. F., & Yuen, A. H. K. (2009). Exploring the effects of gender and learning styles on computer programming performance: Implications for programming pedagogy. *British Journal of Educational Technology*, 40(4), 696–712. <https://doi.org/10.1111/j.1467-8535.2008.00847.x>
- Lau, W. W. F., & Yuen, A. H. K. (2011). The impact of the medium of instruction: The case of teaching and learning of computer programming. *Education and Information Technologies*, 16(2), 183–201. <https://doi.org/10.1007/s10639-009-9118-8>
- Lee, M., & Lee, J. (2021). Enhancing computational thinking skills in informatics in secondary education: The case of South Korea. *Educational Technology Research and Development*, 69(5), 2869–2893. <https://doi.org/10.1007/s11423-021-10035-2>
- Liao, Y.-K. (2000). A meta-analysis of computer programming on cognitive outcomes: An updated synthesis. In J. Bourdeau & R. Heller (Eds.), *Proceedings of ED-MEDIA 2000—world conference on educational multimedia, hypermedia & telecommunications* (pp. 598–604). Association for the Advancement of Computing in Education. <https://www.learntechlib.org/p/16132>
- 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. <https://doi.org/10.2190/e53g-hh8k-ajrr-k69m>

- 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. <https://doi.org/10.1016/j.chb.2014.09.012>
- Mannila, L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). Computational thinking in K-9 education. In A. Clear & R. Lister (Eds.), *Proceedings of the working group reports of the 2014 on innovation & technology in computer science education conference* (pp. 1–29). ACM. <https://doi.org/10.1145/2713609.2713610>
- Mayer, R. E. (1998). Cognitive, metacognitive, and motivational aspects of problem solving. *Instructional Science, 26*(1–2), 49–63. <https://doi.org/10.1023/A:1003088013286>
- Merkouris, A., Chorianopoulos, K., & Kameas, A. (2017). Teaching programming in secondary education through embodied computing platforms: Robotics and wearables. *ACM Transactions on Computing Education, 17*(2), Article 9. <https://doi.org/10.1145/3025013>
- Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *Revista de Educación a Distancia, 46*(10), 1–23. <https://doi.org/10.6018/red/46/10>
- Nardelli, E. (2019). Do we really need computational thinking? *Communications of the ACM, 62*(2), 32–35. <https://doi.org/10.1145/3231587>
- Noh, J., & Lee, J. (2020). Effects of robotics programming on the computational thinking and creativity of elementary school students. *Educational Technology Research and Development, 68*(1), 463–484. <https://doi.org/10.1007/s11423-019-09708-w>
- Organisation for Economic Co-operation and Development. (2013a). *PISA 2012 assessment and analytical framework: Mathematics, reading, science, problem solving and financial literacy*. OECD Publishing. <https://doi.org/10.1787/9789264190511-6-en>
- Organisation for Economic Co-operation and Development. (2013b). PISA 2012 mathematics framework. In *PISA 2012 assessment and analytical framework: Mathematics, reading, science, problem solving and financial literacy* (pp. 23–58). OECD Publishing. <https://doi.org/10.1787/9789264190511-6-en>
- Organisation for Economic Co-operation and Development. (2014). *PISA 2012 technical report*. OECD Publishing. <https://www.oecd.org/pisa/pisaproducts/PISA-2012-technical-report-final.pdf>
- Organization for Economic Co-operation and Development. (2016). *PISA 2015 results (volume i): Excellence and equity in education*. OECD Publishing. <https://doi.org/10.1787/9789264266490-en>
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.
- Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*(2), 137–168. [https://doi.org/10.1016/0732-118X\(84\)90018-7](https://doi.org/10.1016/0732-118X(84)90018-7)
- Pérez-Marín, D., Hijón-Neira, R., Babelo, A., & Pizarro, C. (2020). Can computational thinking be improved by using a methodology based on metaphors and scratch to teach computer programming to children? *Computers in Human Behavior, 105*, 105849. <https://doi.org/10.1016/j.chb.2018.12.027>
- Popat, S., & Starkey, L. (2019). Learning to code or coding to learn? A systematic review. *Computers & Education, 128*, 365–376. <https://doi.org/10.1016/j.compedu.2018.10.005>

- Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the computational thinking test. *Computers in Human Behavior*, 72, 678–691. <https://doi.org/10.1016/j.chb.2016.08.047>
- Saritepeci, M. (2020). Developing computational thinking skills of high school students: Design-based learning activities and programming tasks. *The Asia-Pacific Education Researcher*, 29(1), 35–54. <https://doi.org/10.1007/s40299-019-00480-2>
- Scherer, R., Siddiq, F., & Viveros, B. S. (2019). The cognitive benefits of learning computer programming: A meta-analysis of transfer effects. *Journal of Educational Psychology*, 111(5), 764–792. <https://doi.org/10.1037/edu0000314>
- Selby, C., & Woollard, J. (2013). *Computational thinking: The developing definition*. <https://eprints.soton.ac.uk/356481/>
- Shen, J., Chen, G., Barth-Cohen, L., Jiang, S., & Eltoukhy, M. (2020). Connecting computational thinking in everyday reasoning and programming for elementary school students. *Journal of Research on Technology in Education*, 1–21. <https://doi.org/10.1080/15391523.2020.1834474>
- Shrout, P. E., & Fleiss, J. L. (1979). Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*, 86(2), 420–428. <https://doi.org/10.1037/0033-2909.86.2.420>
- Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, 22, 142–158. <https://doi.org/10.1016/j.edurev.2017.09.003>
- Sullivan, A., & Bers, M. U. (2019). Investigating the use of robotics to increase girls' interest in engineering during early elementary school. *International Journal of Technology and Design Education*, 29(5), 1033–1051. <https://doi.org/10.1007/s10798-018-9483-y>
- Taylor, K., & Baek, Y. (2019). Grouping matters in computational robotic activities. *Computers in Human Behavior*, 93, 99–105. <https://doi.org/10.1016/j.chb.2018.12.010>
- Tedre, M., & Denning, P. J. (2016). The long quest for computational thinking. In J. Sheard & C. S. Montero (Eds.), *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 120–129). ACM. <https://doi.org/10.1145/2999541.2999542>
- Wei, X., Lin, L., Meng, N., Tan, W., Kong, S.-C., & Kinshuk. (2021). The effectiveness of partial pair programming on elementary school students' Computational Thinking skills and self-efficacy. *Computers & Education*, 160, Article 104023. <https://doi.org/10.1016/j.compedu.2020.104023>
- Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25(1), 127–147. <https://doi.org/10.1007/s10956-015-9581-5>
- Wing, J. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35. <https://doi.org/10.1145/1118178.1118215>

- Witherspoon, E., Higashi, R., Schunn, C., Bachr, E., & Shoop, R. (2017). Developing computational thinking through a virtual robotics programming curriculum. *ACM Transactions on Computing Education*, 18(1), Article 4. <https://doi.org/10.1145/3104982>
- Zhao, L., Liu, X., Wang, C., & Su, Y.-S. (2022). Effect of different mind mapping approaches on primary school students' computational thinking skills during visual programming learning. *Computers & Education*, 181, Article 104445. <https://doi.org/10.1016/j.compedu.2022.104445>

Author Biography

Ndudi O. Ezeamuzie is a doctoral student in the Faculty of Education, The University of Hong Kong. He is also a computer scientist with research interest in computational thinking and programming in the context of STEM education.