Relationships between computational thinking and the quality of computer programs

Kay-Dennis Boom, Matt Bower, Jens Siemon, Amaël Arguel

Education and Information Technologies
https://doi.org/10.1007/s10639-022-10921-z

Relationships between computational thinking and the quality of computer programs

Kay-Dennis Boom[1] · **Matt Bower**[2] · **Jens Siemon**[1] · **Amaël Arguel**[3]

Received: 5 September 2021 / Accepted: 27 January 2022

Abstract

Computational thinking – the ability to reformulate and solve problems in ways that can be undertaken by computers – has been heralded as a foundational capability for the 21st Century. However, there are potentially different ways to conceptualise and measure computational thinking, for instance, as generalized problem solving capabilities or as applied practice during computer programming tasks, and there is little evidence to substantiate whether higher computational thinking capabilities using either of these measures result in better quality computer programs. This study examines the relationship between different forms of computational thinking and two different measures of programming quality for a group of 37 pairs of pre-service teachers. General computational thinking capabilities were measured using Bebras tests, while applied computational thinking processes were measured using a Computational Thinking Behavioural Scheme. The quality of computer programs was measured using a qualitative rubric, and programs were also assessed using the *Dr Scratch* auto-grading platform. The Test of Nonverbal Intelligence (3rd edition, TONI-3) was used to test for confounding effects. While significant correlations between both measures of computational thinking and program quality were detected, regression analysis revealed that only applied computational thinking processes significantly predicted program quality (general computational thinking capability and non-verbal intelligence were not significant predictors). The results highlight the importance of students developing applied computational thinking procedural capabilities more than generalized computational thinking capabilities in order to improve the quality of their computer programs.

Keywords Computational thinking · Visual programming · Scratch · Program quality

Extended author information available on the last page of the article

Published online: 03 March 2022

1 Introduction

1.1 Context of the problem

Since its first major appearance in 2006 by Wing, computational thinking has been intensively discussed in the field of computer science education (Tang, Chou, & Tsai, 2020). CT can be regarded as the ability to reformulate problems in ways that computers can then be used to help solve those problems (International Society for Technology in Education [ISTE] & the Computer Science Teachers Association [CSTA], 2011). The value proposition of computational thinking capabilities in a digital age is that they can help people solve a range of problems that lead to personal satisfaction and success, not only in the technology area but also life more broadly. However, the conjecture that possessing computational thinking knowledge, or applying computational thinking skills while solving problems, leads to higher quality solutions, has rarely been empirically validated.

One aspect of computational thinking that is often emphasized by advocates is that it is not simply computer programming capability. Research about the effects of computational thinking knowledge and/or skill can be divided into the area of effects regarding computational problem solving (e.g. computer programming) and effects regarding diverse non-programming problems or tasks. For example, a wide range of problems, from finding the shortest route between map locations to designing an online shopping platforms, rely on people applying computational thinking processes while they are writing computer programs to solve those problems. However, computational thinking skills (such as problem decomposition, pattern recognition, algorithmic thinking and abstraction) can also be used to solve a range problems that do not involve computer programming, such as finding a way through a maze or specifying the steps in a dance sequence. While learning computer programming relies on people utilizing and applying computational thinking as part of the process they undertake, instructional settings will often use computational thinking foundations to teach subjects and ideas that do not involve computer programming (e.g. Bull, Garofalo, & Hguyen, 2020). In fact, a literature review conducted by Tang et al. (2020) concluded that there were far more computational thinking effects analyzed in subject areas not related to computer science ($n = 240$) than for effects related to computer science ($n = 78$). However, we note that while computational thinking can be applied in a range of disciplines, it is considered absolutely essential and fundamental to successful computer programming (Angeli & Giannakos, 2020; Lye & Koh, 2014). Yet, we could not find any studies amongst the literature that examined whether or not computational thinking capabilities did in fact relate to higher quality computer programs.

The purpose of this study is to evaluate the extent to which general computational thinking knowledge, as well as computational thinking processes applied during problem solving tasks, influence the quality of computer-programming solutions. This was achieved by comparing university students' computational knowledge (as measured by Bebras tests) and the computational thinking processes observed while they wrote computer programs with the quality of the final computing products that

they produced. The findings of this study have implications for how computational thinking is framed, conceptualized and emphasized within education and society.

2 Literature review

2.1 Defining computational thinking and its subcomponents

Computational thinking is generally seen as an attitude and skill for solving problems, designing complex systems, and understanding human thoughts and behaviors, based on concepts fundamental to computer science (Lye & Koh, 2014). Recent reviews of computational thinking definitions and components by Shute, Sun and Asbell-Clarke (2017) and Ezeamuzie and Leung (2021) point out the lack of consistent definition regarding what is meant by computational thinking, though with some terms being more popular (such as abstraction, algorithm design, decomposition, and pattern recognition as generalisation), particularly when academics devise explicit definitions with relation to their research. Some inconsistency between definitions of components can occur, at times not because there is disagreement about what computational thinking involves, but because other frequently used terms such as 'sequencing', 'conditional logic' and 'loops' can conceptually fall within overarching categories (in this case, 'algorithm design').

In this study, we will draw upon generally accepted core components of computational thinking as being comprised of problem decomposition, pattern recognition (generalisation) algorithmic thinking and abstraction, which accords with other definitional work from the research field (Angeli & Giannakos, 2020; Cansu & Cansu, 2019; Tsai, Liang, Lee, & Hsu, 2021). We acknowledge that there are other aspects of computational thinking that are identified in some studies, such as 'parallelism', 'data collection', and 'modelling', as outlined by Shute et al. (2017), however, as Ezeamuzie and Leung (2021) points out, these sorts of other terms are relatively uncommon, and they are not processes utilised for all computational thinking problems. Selecting problem decomposition, pattern recognition (as generalisation), algorithmic thinking and abstraction as the components of computational thinking in this study also corresponds with approaches adopted in industry (for example, Csizmadia, Curzon, Dorling, Humphreys, Ng, Selby, & Woollard, 2015; McNicholl, 2019).

One crucial part of any computational thinking task is *problem decomposition*, the division of a problem into smaller chunks. Problem decomposition has been identified as a general problem solving strategy well before the advent of computational thinking (Anderson, 2015). In computational problems, decomposition is particularly important because of its relationship to modularity, where the complexity of a task can be simplified by identifying smaller parts that can each be addressed separately (Atmatzidou & Demetriadis, 2016). For example, when programming a multimedia story, one might first identify the different scenes that occur, and then break each scene into a series of actions by the characters.

Another component of computational thinking is *abstraction*, in terms of ignoring unimportant details and instead focusing on relevant information. From a psychological perspective, abstraction is a thought process that is used to achieve organised

thinking (Shivhare & Kumar, 2016). In computational problems, abstraction enables people to concentrate on the essential, relevant, and important parts of the context and solution (Thalheim, 2009). For instance, when writing a multimedia story to have characters dance about a screen, a person may recognise that their program only needs to attend to the coordinates of the characters and their size, and the routines that they write can be applied to numerous characters irrespective of their colours or costumes.

A further critical thought process when engaging in computational thinking is *pattern recognition*. Pattern recognition involves being able to infer rules based on observations and apply these rules to instances that have never been encountered (visa vi Posner & Keele, 1968). Pattern recognition is crucial when solving computational problems, because rules inferred based on observations can then be translated into instructions that can be used to solve problems. For instance, when a person realises that a square can be drawn by drawing a straight line and then turning 90 degrees four times, then they can easily and efficiently specify a set of instructions for a computer (or human) to execute the process. It is important to note that pattern recognition is closely related to abstraction as a form of ignoring irrelevant details, but is generally regarded as distinct by virtue of distilling those aspects of a situation that repeat or reoccur in certain ways.

The fourth computational thinking category in this study is *algorithmic thinking*. An algorithm is a well-defined procedure or 'recipe' that defines how inputs can be used to achieve a specific objective (Cormen et al, 2014; Sipser, 2013). Algorithmic thinking has roots in cognitive psychology in the form of scripts, that help people to know how to behave in social or behavioural contexts (for instance, going to a restaurant or playing a game, see Schank & Abelson, 1977). When solving computational problems, algorithmic thinking enables people to translate their abstract ideas and the patterns that they recognise into a set of procedures, for instance having a robot trace out a square and then dance on the spot. For the purposes of this study, algorithmic thinking also includes the thinking required to resolve errors that occur in early versions of algorithm designs (the process known in computing as 'debugging'), thus overcoming issues associated with delineating these two intrinsically interrelated processes.

2.2 Ways of measuring computational thinking

When defining a skill, the question arises whether it is possible to measure and differentiate it from other, possibly overlapping or more general skills. For computational thinking, the existing measurement methods that can be broadly divided into assessment of computational thinking as knowledge that is applied or tested (input), the assessment of computational thinking as a skill observed during a problem solving activity (process) and (theoretically also) the assessment of computations thinking by analyzing the result of a task (output). All measures are subsequently used as indicators for the existence and the grade/level of the respective type of the computational thinking competence.

The most internationally well-known instruments for measuring general computational thinking knowledge are the Bebras Challenges. The main idea behind Bebras

Challenges has been to create abstract, non-computing problems that require specific cognitive abilities rather than technical knowledge or coding experience (Dagienė & Sentance, 2016). Examples of Bebras tasks can be found at https://www.bebras.org/examples.html. Different studies have shown that abilities such as breaking down problems into parts, interpreting patterns and models, and designing and implementing algorithms are needed to solve Bebras problems (Lockwood & Mooney, 2018; Araujo, Andrade, Guerrero, & Melo, 2019). There are also other approaches to measuring general computational thinking knowledge, both within computer programming contexts and also other disciplinary contexts, many of which have been applied in the training of teachers. For instance, Zha, Jin, Moore, and Gaston (2020) used multiple choice knowledge quizzes about computational thinking and Hopscotch coding to measure the impact of a team-based and flipped learning introduction to the Hopscotch block coding platform. In a study exploring the effects of a 13 week algorithm education course on 24 preservice teachers, Türker & Pala (2020) used the "Computational Thinking Skills Scale" (CTSS, from Korucu, Gencturk & Gundogdu, 2017) comprising the computational thinking facets creativity, algorithmic thinking, collaboration, critical thinking and problem solving. Suters and Suters (2020) report on a paper-and-pencil based computational thinking knowledge assessment to measure the effects of an extend summer institute for middle school mathematics teachers (n = 22) undertaking training in computer programming with Bootstrap Algebra and Lego® Mindstorms® robotics. The content assessment consisted of items that integrated mathematics common core content with facets of computational thinking, in line with research endeavors recognizing the need to contextualize computational thinking within specific disciplines (Gadanidis, 2017; Grover & Pea, 2013; Weintrop et al., 2016). All of these approaches to computational thinking knowledge assessment share an emphasis on short, often multiple choice, closed questioning to measure computational thinking, rather than examining the computational thinking that arises as part of authentic and more extended problem solving contexts.

In a second variant of possible computational thinking measurement, the process of solving a context-dependent task – mostly typically a programming task – is observed and analyzed with regard to the abilities which are considered to be part of computational thinking skill. Skill analysis based on observations is a comparatively underdeveloped field. Brennan & Resnick (2012) seminally investigated the computational thinking processes and practices that children undertook while designing their programs using the visual programming platform Scratch, noting that "framing computational thinking solely around *concepts* insufficiently represented other elements of students learning" (p. 6). Their qualitative observations and interviews identified computational thinking practices such as being incremental and iterative, testing and debugging, reusing and remixing, and abstracting and modularizing. However, their results were not reported based on any sort of observational coding of participants, so that there is no indication of time spent on each of these processes while solving computing problems.

While analysis of learner pre- and/or post- interview narratives has been previously conducted to determine evidence of computational thinking (Grover, 2011; Portelance & Bers, 2015), we were not able to find any computational thinking analyses involving systematic examination of narratives emerging from participants while

they were solving authentic programming problems. However, there are examples of observing and thematically categorizing computer programming processes and narratives (Bower & Hedberg, 2010; Knobelsdorf & Frede, 2016). These approaches provide a basis for in-situ observation and subsequent qualitative analysis of programming activity for computational thinking constructs such as problem deconstruction, abstraction, pattern recognition and algorithmic thinking, and our study is based on these more systematic observational approaches.

2.3 Measuring the quality of computer programs

There are a range of qualities that can be used to evaluate the quality of computer programs, such as the extent to which the code functionally achieves its intentions, avoids unnecessary repetition, is well organized, and so on (Martin, 2009). Much of the research relating to evaluating the quality of computer programs examines how to ways of auto-assessing student work (for instance, Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010; Pieterse, 2013). However, automated tools struggle to accurately assess computational thinking (Poulakis & Politis, 2021), and recent work points out the need to look beyond raw functionality and 'black-box' testing of outputs, to examine the inner working of code and algorithms (Jin & Charpentia, 2020). Some research also examines the extent to which computational thinking is evident with the final programming product itself, by virtue of the code fragments that are used and their sophistication. Brennan and Resnick (2012) examined whether aspects of computational thinking were present in students' block-based Scratch programs. Grover et al. have manually evaluated computational thinking evident in students' Scratch programs, though without providing detail of the process and rubrics (Grover, 2017; Grover, Pea, & Cooper, 2015). An increasingly renown innovation, Dr Scratch, combines automated assessment, examination of the inner workings of programs, and analysis of computational thinking to provide a measure of program quality for Scratch programs (Moreno-León & Robles, 2015). One study has established a strong correlation ($r = 0.682$) between the Dr Scratch automated assessment of computational thinking evident within students' Scratch programs and manual evaluation of computational thinking within Scratch programs by human experts (Moreno-León et al., 2017). However, the computational thinking within a computer program is not necessarily a proxy for overall program quality, and the extent to which program quality relates to the computational thinking knowledge and computational thinking processes of program authors is an open question.

2.4 Research question

Thus, having established the lack of empirical evidence to suggest that general computational thinking knowledge or in-situ computational thinking processes is related to computing performance, and armed with potential ways to operationalize and measure computational thinking knowledge, computational thinking processes, and quality of computer programs, this study examines the following research questions:

1. Is the quality of computer programming solutions that people produce related to their general computational thinking knowledge?
2. Is the quality of computer programming solutions that people produce related to the applied computational thinking processes that they undertake?

3 Method

3.1 Participants

The sample for this study was drawn from 74 pre-service teachers completing a digital creativity and learning course at an Australian university. Among them 68% were female, 30% male and 2% preferred not to say. On average, participants were 23.9 years old ($SD = 5.2$). In terms of language proficiency, 97% indicated that they spoke English fluently or were native speakers. In terms of prior knowledge, 97% had no or only little prior programming experience and none of the participants were familiar with the Scratch programming environment that was used for the study.

3.2 Instruments

3.2.1 Measuring computational thinking knowledge

To measure computational thinking knowledge as it arises in general problem solving contexts, participants solved an online version of adapted Bebras tasks. All tasks were chosen from the Australian versions of the Bebras contests from 2014 (Schulz & Hobson, 2015) and 2015 (Schulz, Hobson, & Zagami, 2016). Only tasks from the oldest available age group were selected (i.e., for adolescents 16 to 18 years of age and school levels 11 and 12, respectively). The tasks were slightly revised and presented without any iconic beavers or other comical pictures in order to be more appropriate for the university participants in this study. Although there is still a considerably age gap between the targeted age group of the tasks and the actual age of participants, it was not expected that this difference would cause any problems (e.g., ceiling effects) because pre-service teachers on the whole were not expected to be familiar with or particularly adept at computational thinking tasks. The scoring of participant performance was based on the recommended scoring system of the founder of the Australian version of the Bebras tasks (Schulz et al., 2016). There were eight tasks considered as easy level (worth two points), seven medium (three points), and five hard tasks (four points) resulting in 20 tasks in total with a maximum achievable score of 57.

3.2.2 Observing computational thinking processes

To enable participants to demonstrate how much time they spent on computational thinking-relevant processes while programming, participants were set a Scratch programming task. Scratch itself was developed in 2003 at MIT Media Laboratory and publicly launched in 2007 (Resnick et al., 2009). It is one the first and one of the

most popular open-source visual programming environments. In visual programming environments, users connects code blocks with each other instead of actual writing a code as common in real programming languages.

To prepare students for the task, they were given 45 min to review the Scratch tutorials available from within the Scratch platform. They were also allowed to access these tutorials during the programming task. The task itself was defined as follows: "*Program a story or a game where a hero has to overcome a challenge in order to defeat the villain(s)*."

This task was chosen because it is somewhat open-ended and can be solved in the chosen Scratch development framework without prior programming knowledge. Furthermore, computational thinking subskills (problem decomposition, pattern recognition, abstraction, algorithm design) would most likely have to be used to solve the task. The way in which the Scratch programming environment, task, and participants may influence the generalizability of results is considered in the Discussion section of this paper.

To reliably assess the amount of time they spent on computational thinking processes during their Scratch programming session, a computational thinking behavior scheme (CTBS) was developed. The CTBS was based on event sampling, involving analysis of how often and for how long specific behavioral cues occur. Based on the literature review, four components were identified as main features of computational thinking and which are the latent constructs in the CTBS: decomposition, abstraction (as in ignoring unimportant details), pattern recognition, and designing and applying algorithms (see operationalization of these constructs in Table 1 below). Two

Table 1 Operationalization of computational thinking constructs in the Computational Thinking Behavioural Scheme (CTBS)

Computational thinking components (latent variables)	Behavioural indicator (manifest variables)
Decomposition	Put problem into pieces / building sub tasks or problems Identifying the immediate next step Discussing if then relations of the story or game (is related to programming elements)
Abstraction	Focusing on important information; neglecting unimportant details Simplifying anything (problem, sub problem, functions, code bocks, etc.)
Pattern recognition	Identifying similar characteristics (sub problems, functions, code blocks, etc.) Use of copy-paste Aha moments (must be related to an event when student understood relationship between things)
Algorithm design	Putting code chunks together Testing and judging algorithm (i.e., clicking on run or double click on sequence or actively observing a running sequence) Debugging - try to find error and adjust algorithm

researchers coded five entire videos independently to assess inter-rater reliability. As a result, at least two third of the events were identified by both raters. The range of the frequency of agreement for the five videos laid between 66.67% and 72.50% and the κ coefficients ranged from 0.58 to 0.67. Overall, the reliability can be interpreted as moderate (Landis & Koch, 1977). Note that the CTBS measured the time spent on computational thinking components, not the correctness of the computational thinking processes. It is to be expected that during the process of solving computational problems that people may not always immediately have correct thoughts about the right course of action, and this study sought to examine relationship between the time spent on computational thinking processes and the quality of programming products.

3.2.3 Measuring program quality

To measure participants' program quality, two measures were used. For one, a rubric scheme loosely based on "Clean code" of Martin (2009) was developed specially for this study. The program quality criteria were based on five categories: *richness of project*, *variety of code usage*, *organization and tidiness*, *functionality of code* and *coding efficiency*. Richness of project described how much was happening in the Scratch project. Lower scores were given when only one element was programmed to perform only one behaviour, while Scratch projects consisting of several programmed elements that were related to each other received higher scores. The variety of code usage depended on the kinds of code blocks were used. Scratch projects were rated lower when they mainly consisted of simple code chunks such as *motion* or *looks* and high when more advanced chunks like *control* or *sensing* were used. The category organization and tidiness took into account the extent to which the control section in Scratch was organized, with more organized Scratch projects receiving higher scores. Functionality was assessed based on whether the intention of the Scratch project was clear and whether it worked as intended. Projects received higher scores when they ran smoothly and the intention was easy to understand. The category efficiency

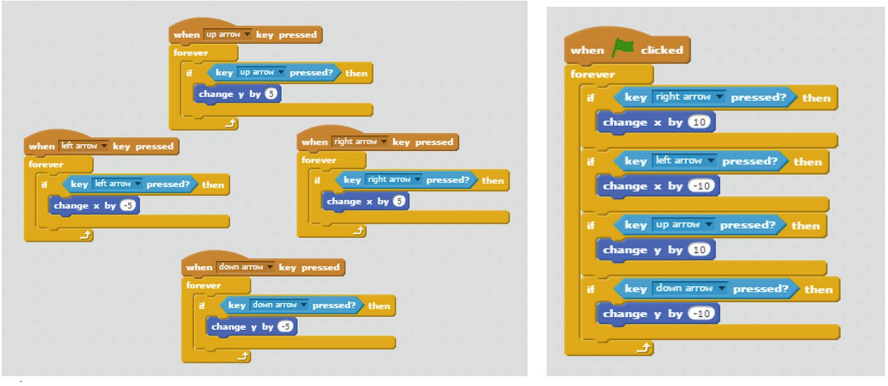

Figure 1 Two examples of the same function but coded differently. An example with unnecessary duplicates is shown on the left and a more efficient version is seen on the right

described the usage of code controlling the flow of execution, and the number unnecessary duplications. Lower scores were given to projects having many such duplicates, while more generalized and more abstract code scripts received higher scores. An example of a program with unnecessary duplication is shown in Fig. 1 (left), compared to a more efficiently represented code block in Fig. 1 (right).

The five code quality categories were all rated on a scale including 0 (*not evident*), 1 (*poor*), 2 (*satisfactory*), 3 (*good*), up to 4 (*excellent*). A weighted mean over all categories was calculated to provide a general assessment. The weight for each category was based on their importance for program quality, resulting in extent and richness, variety, and functionality being weighted 20% each, efficiency 30%, and organization and tidiness 10% to the weighted mean. Quality criteria and the (weighted) scoring system of the scheme were discussed with two computer science education professionals to uphold the content validity of the measure. In addition, one of the CS education professionals rated the Scratch projects to obtain reliability assessment. Inter-rater reliability was high with ICC(3,1), 95% CI [0.87, 0.96].

The second measure for program quality was based on Dr Scratch (Moreno-León & Robles, 2015). Dr Scratch provides a measure of program quality based on seven dimensions relevant to CS: abstraction and problem decomposition, parallelism, logical thinking, synchronization, algorithmic notions of flow control, user interactivity and data representation. Dimensions are judged as 0 (*not evident*), 1 (*Basic*), 2 (*Developing*), and 3 (*Proficient*). Scores are aggregated over all dimensions resulting in a total evaluation score (mastery score) from 0 to 21. Mastery scores between 8 and 14 are regarded as *general developing*; lower than 8 is regarded as *generally basic*, and more than 14 as *general proficient*. High correlations between Dr Scratch mastery scores and experts judgments of program quality can be used as an indicator of satisfactory criterion validity (Moreno-León, Román-González, Harteveld, & Robles, 2017). While Dr Scratch focuses primarily on computational thinking elements as opposed to other aspects of computer programming (e.g. organization of code, efficiency), it is based on the final computer programming solution that is produced, and thus provides an interesting alternative measure of program quality for this computational thinking study.

3.2.4 Test of nonverbal intelligence

To take account for potential confounding effect, participants' nonverbal intelligence was also measured. For this the Test of Nonverbal Intelligence (3rd edition; TONI-3, Brown, Sherbeernou, & Johnson, 1997) was used. The TONI-3 is a classic culture fair test (i.e., minimally linguistically demanding) and as in many of them participants need to recognize a correct figure in a set of abstract and geometrical pictures. The test consists of 45 items and has an average testing time of 15 min and has a satisfactory level of psychometrical properties (Banks & Franzen, 2010).

3.3 Procedure

Initially, participants completed the Bebras Computational Thinking Knowledge test and the test of nonverbal intelligence (TONI-3 online). One week later, the second

phase took place at university's classrooms and participants attempted the task in Scratch. To collect rich video material with many verbal and nonverbal indicators for the research team to analyze, participants were organized in pairs. It was hoped that working in pairs would encourage participants to talk and engage more with each other. The pairs were formed based on similar Bebras scores to minimize any effects due to large differences in competences. In total, 37 pairs were formed and filmed while working on the task, forming the corpus for the analysis.

3.4 Analysis

All statistical analysis was conducted using the R statistics programming environment. In order to acquire a sense of the data, basic descriptive statistics including means and standard deviations were calculated for all five measures (Bebras scores, time spent on different computational thinking processes, program quality rubric score, Dr Scratch score, TONI-3 non-verbal intelligence score). Because participants worked on the programming task in pairs, all programming assessments based on the rubric scheme, the additional Scratch evaluation assessment based on Dr Scratch, and the assessment of how much time participants spent on computational thinking behavior based on CTBS, were paired values. Scores on the Bebras tasks and TONI-3 test were averaged for each pair. Of the 37 pairs of participants who agreed to complete the Bebras test and have their final programs used in the study, 27 agreed to be video recorded for the purposes of the CTBS analysis, and 32 pairs agreed to complete the TONI-3 test.

Spearman's ρ were computed between all five measures using all available data, to determine whether the underlying variables were directly correlated. Finally, in order to account for the possibility of moderating variables, two regression models were estimated with the two program quality measures as outcomes (program quality rubric score and Dr Scratch score). These two regression models used the Bebras task scores, the CTBS, and the TONI-3 IQ scores as predictors, so that it was possible to detect if any of these were moderating variables.

4 Results

4.1 Descriptive statistics and measurement outcomes

The average score for the measure of general computational thinking capability (Bebras task) was 57.03% ($SD = 18.6\%$). The range was from 21% to one participant who achieved 100%. Results indicated a medium level of test difficulty, with no serious problems due to ceiling or floor effects. In TONI-3-IQ, participants achieved an average intelligence score of 113.12 ($SD = 14.17$). The mean of this sample was slightly higher than the expected value of the population ($\mu = 100$, see, for example, Sternberg, 2017), which can be explained by the fact the sample was drawn from university students. The time participants needed to complete the Bebras tasks ($Md = 55$ min) and the TONI-3-IQ ($Md = 22$ min) roughly aligned with the expected time of 60 min (Dagienė & Futschek, 2008) and 15 min (Brown et al., 1997), respectively.

Table 2 shows that while writing their programs, coded participants spent nearly half of their time on computational thinking behaviors, with algorithmic design having the largest contribution and little time spent on decomposition and pattern recognition. Pattern recognition was observed in less than half of all pairs. No sign of abstraction in the sense of neglecting information was observed for any pair.

Table 3 contains an overview of scores achieved by the pairs of participants in the rubric scheme for program quality. The full range of the rating scales (0 to 4) was used. The distributions of all five dimensions had their center at around 2 (i.e., *satisfactory* level).

Table 2 Overview of Coded Events and Time Spent on Computational Thinking Behavior

Computational thinking component	Pairs	events	% of time spent on CT-relevant behavior	
			M (*SD*)	*Max* - *Min*
Decomposition	27	310	7.77 (5.35)	22.61 – 1.03
Abstraction	-	-	-	-
Pattern recognition	17	53	1.43 (1.05)	3.75 – 0.18
Algorithmic design	27	1,072	37.46 (12.26)	61.06 – 10.39
Computational thinking overall	27	1435	46.66 (14.96)	70.42 – 15.74

Table 3 Overview of Rubric Scheme for Programming Quality

Programming dimensions	*M* (*SD*)
Extension	1.86 (0.89)
Variety	2.19 (1.02)
Organization	1.84 (0.87)
Functionality	1.92 (0.95)
Efficiency	2.08 (1.21)
Weighted mean	2.00 (0.91)

Note: pairs = 37

Table 4 Overview of Dr Scratch Project Scores

Dr Scratch dimension	Absolute frequency of level				*M* (*SD*)	*Mdn*
	0	1	2	3		
Abstraction and problem decomposition	2	35	-	-	0.95 (0.23)	1
Parallelism	5	21	4	7	1.35 (0.95)	1
Logical thinking	15	20	2	-	0.65 (0.59)	1
Synchronization	14	11	1	11	1.24 (1.26)	1
Flow control	-	9	28	-	1.76 (0.43)	2
User interactivity	-	11	25	1	1.73 (0.51)	2
Data representation	1	22	14	-	1.35 (0.54)	1

In comparison, participants' Scratch projects typically only achieved a *basic* rating according to Dr Scratch, with only two dimensions rated as *developing* (See Table 4).

4.2 Correlations between variables

As a first step towards analyzing which of the two computational thinking measures (general computational thinking knowledge as measured by Bebras versus computational thinking processes as observed in practice) have a greater relationship to program quality, Spearman's ρ were computed (see Table 4). Correlation between the two measures of program quality (weighted means based on the developed rubric scheme and Dr Scratch mastery scores) revealed a significant relationship, $\rho = 0.61$, $p < .001$. Based on common interpretation of effect sizes (Cohen, 1988), this correlation can be interpreted as large. The large correlation between the two measures of program quality reveals a degree of consistency in their assessment of student programs.

Significant positive correlations were found between general computational thinking knowledge (Bebras scores) and both measures of program quality, with a borderline small to medium effect sizes (see Table 5). Significant positive correlations between time spent on computational thinking processes while programming and both measures of program quality, with quite large effects.

Because of some potential (partial) conceptual overlaps between nonverbal intelligence and computational thinking, the correlations between the TONI-3 IQ and computational thinking measures were calculated as well. On one hand, the correlation between the TONI-3 IQ and the Bebras scores was significant and positive with a medium to large effect size, $\rho = 0.49$, $p = 0.002$. On the other hand, correlation between TONI-3-IQ and time spent on computational thinking processes while programming was not statistically significant, $\rho = 0.09$, $p = 0.346$

4.3 Regression analysis

As explained in the Methodology section, two regression models were estimated with both program quality measures as outcomes and the both computational thinking measures and the TONI-3 IQ scores as predictors. Standardized parameter estimations and tests of significance of the regression model are shown in Table 6. The regression models only partly supported the findings from the correlations. The pos-

Table 5 Spearman' ρ Correlations Between Programming Quality, Dr Scratch and Different Measures

	Programming rubric scheme		Dr Scratch mastery score		*N* (pairs)
	ρ	*p*	ρ	*p*	
Bebras score	0.32	0.027	0.29	0.041	37
Time of computational thinking behavior	0.65	< 0.001	0.57	0.001	27
IQ based on TONI-3	0.29	0.055	0.19	0.149	32

Note: one-sided *p*-values

Table 6 Regression Models

	Program quality			Dr Scratch mastery score		
Predictors	β	*t*-value (*SE*)	*p*	β	*t*-value (*SE*)	*p*
Bebras score	–0.41	–1.95 (1.24)	0.066	–0.14	–0.62 (4.27)	0.542
Time on computational thinking behavior (overall)	0.74	4.31 (0.01)	< 0.001	0.70	3.86 (0.03)	< 0.001
TONI-3 IQ	0.36	1.82 (0.01)	0.084	0.11	0.53 (0.05)	0.599
R^2 (R^2_{adj})	0.50 (0.42)			0.44 (0.36)		
F(3,20)	6.60		0.003	5.29		0.008

Note: N = 24. The intercept is omitted for better overview

itive correlation between the Bebras score and both measures of program quality vanished when taking into account the effect of TONI-3 IQ. The only significant predictor for both measures of programming quality was the computational thinking process scores.

Post hoc analyses for both regression models were performed for power estimation. Based on the given parameters ($N = 24$, number of predictors = 3, effect size = $R^2_{pro.qual} = 0.50$, $R^2_{DrScratch} = 0.44$, and $\alpha = 0.05$), a power of > 0.99 for both models was achieved. Because of the small sample size, assumptions about linear multiple regressions such as homoscedasticity, multicollinearity, and residuals were rigorously checked. No serious violations of any assumption could be found, though it should be noted that the residuals when the outcome was programming quality were not normally distributed, based on Shapiro-Francias test, with $W' = 0.88$, $p = 0.011$. In conclusion, the power of both regression models were sufficiently high enough and the regression coefficients can be interpreted as "best linear regression estimations".

5 Discussion

The general computational thinking knowledge scores (Bebras) and the computational thinking procedural performance (as indicated by the CTBS), were both positively correlated with both program quality measures (the rubric scheme and Dr Scratch mastery score). Therefore, a general interpretation could be that the higher the level of both general computational thinking knowledge and applied computational thinking in practice, the better the program quality. However, this interpretation would be premature because regression analyses revealed that only one — the applied computational thinking in practice — was a significant predictor of program quality when controlling for other variables such as the level of nonverbal intelligence and general computational thinking knowledge. The reason why the two different computational thinking measures predict programming differently might lie in different perspectives underlying the two different measures of computational thinking, and how these might mediate the relationship with program quality.

The Bebras tasks focus on general and conceptual aspects of computational thinking. Correlations between the Bebras score and the TONI-3-IQ were between moderate and strong. As for the most instruments for nonverbal intelligence, TONI-3 is based on pictures in which participants need to identify similar instances and recog-

nize patterns. Many of the Bebras tasks are designed in a similar fashion. The original idea behind the Bebras tasks was to create a test about CS concepts "independent from specific systems" to avoid contestants being dependent on prior knowledge of any specific IT system (Dagienė & Futschek, 2008, p. 22). This may have led to some items being similar to those of nonverbal intelligence tests.

As found in some prior studies, this also caused confusion for some Bebras contestants. Vaníček (2014) asked participants for their opinions about the Bebras tasks. Some questioned the purpose and validity of the test, stating "I wonder what the contest questions have to do with informatics. Maybe nothing at all?". If (at least some) Bebras tasks are similar to those of nonverbal intelligence tests and there is a high and significant positive correlation between both measures, it is possible that both tests measure similar constructs. This would explain why the relationship between the Bebras scores and program quality vanished when controlled for TONI-3-IQ. The Bebras tasks are validated by several studies (Dagienė & Stupuriene, 2016; Dolgopolovas, Jevsikova, Savulionienė, & Dagienė, 2015; Lockwood & Mooney, 2018) but none of these studies controlled for any potential confounding effects on similar psychological constructs such as nonverbal intelligence. We could only find one study in which the potential relationship between the Bebras tasks and nonverbal intelligence has been discussed, with similar findings to our study (Román-González, Pérez-González, & Jiménez-Fernández, 2017). Thus, it is possible that the Bebras tasks indeed measure computational thinking but mainly the facet of abstract thinking and pattern recognition.

It is possible that these abstract parts of computational thinking alone are not a good predictor of programming quality because extensive cognitive effort is required to transfer the skills for application in different situations and settings. Even though some similar skills are required to solve both kinds of tasks (the Bebras tasks as well as the programming task in this study), it would require a high level of transferability from these abstract logical quizzes to real applied programming situations. Moreover, according to the authors of the Bebras tasks, participants need to apply the same cognitive abilities as needed for programming tasks such as problem deconstruction, thinking abstractly, recognizing patterns, and being able to understand, design, and evaluate algorithms (Dagienė & Sentance, 2016). However, the content of the Bebras tasks (as for many 'unplugged' methods) is very different from real programming tasks. This may lead to general computational thinking as measured by Bebras tasks not providing a good predictor of program quality above and beyond that which is captured and controlled for by general measures of intelligence (such as the TONI-3-IQ).

In our opinion, the results can be well explained in terms of the thesis of disproportion between application extensity and intensity of use (Weinert, 1994). This theory asserts that, the more general a rule or strategy is, the more minor its contribution to solving challenging, content-specific problems. This could also apply to the computational thinking skills of the Bebras tasks. The measured skills are very general and partly overlap with general facets of intelligence. Their contribution to solving a challenging, content-specific problem might therefore be rather small and statistically hard to detect. At least, this would be one possible interpretation of the rather

weak correlation and the lost connection in the regression analysis with regard to general computational thinking knowledge and program quality.

In contrast to the Bebras tasks, the focus of the CTBS lies on participants' applied computational thinking processes in practice. Correlations indicated that the more participants spent on applied computational thinking processes, the better the programming quality of their Scratch project. It must be pointed that this was mostly due to algorithmic design, with algorithmic design being the most frequently applied computational thinking activity measured. As stated before, participants were working on their code from the start of the session and so there is a logical interpretation that the longer and the more participants spent on algorithmic design, the better the quality of their programs. Even after controlling for other measures, this relationship was still significant and persisted in both regression models with the programming quality rubric and Dr Scratch project evaluation as outcome, respectively. What is even more remarkable is that computational thinking processes were significantly correlated with program quality even though the correctness of the computational thinking processes was not assessed in this study. That is to say, that the more time spent thinking about computational thinking components while solving the computing problem led to better quality programming solutions, even when at times that computational thinking may not have necessarily been 'right'. This is in line with the learning concept of 'productive failure', where thinking deeply about problems and exploring incorrect solutions can ultimately lead to greater learning overall (Kafai, De Liema, Fields, Lewandowski & Lewis, 2019).

These results indicate that the computational thinking process capabilities observed by the CTBS are more strongly related to program quality than computational thinking knowledge as measured by Bebras. While the Bebras Challenge is undoubtedly a valuable competition for students worldwide, the results from this study indicate the ability to solve Bebras problems may not be a good indicator of the ability to solve authentic informatics problems that involve computer programming. In fact, the result challenges the premise that generalised computational thinking knowledge underpins the ability to solve authentic programming problems to any substantial extent. The capacity to apply computational thinking processes in-situ has been shown in this study to be far more relevant and influential in terms of being able to derive high quality programming solutions than solving general computational thinking knowledge problems. To this extent, from a pedagogical perspective, educators who wish to use computational thinking as a basis for improving the ability of their students to solve programming problems should focus on developing students' abilities to apply computational thinking processes in practice (algorithm design, problem decomposition, pattern recognition, abstraction) rather than their computational thinking knowledge in a more detached and decontextualized sense.

5.1 Limitations of the Study

In this study, students worked together in pairs as a naturalistic way to provoke social interaction and make otherwise unobservable thoughts accessible. This contributed to the authenticity of the study, with pair programming often occurring in industry and education. Moreover, pair-programming settings have been used in prior stud-

ies in terms of measuring computational thinking and programming knowledge for novices (Denner, Werner, Campe, & Ortiz, 2014; Wu, Hu, Ruis, & Wang, 2019). However, this approach involved some inherent challenges. It was not possible to perfectly group pairs according to identical levels of computational thinking, intelligence, or programming quality. Some might argue that the results and overall conclusion might have been different if all measures were obtained and analyzed solely on an individual basis. However, gauging individual measures of computational thinking programming skills from a behavioral perspective also involves challenges, as it is difficult to encourage individual participants to verbalize their thinking for the entire duration of the programming process. We believe that the benefits of analyzing computational thinking arising from a more naturalistic setting outweigh those from measuring the computational thinking of individuals, in terms of the validity of results.

It is also worth mentioning that the CTBS and the programming quality instrument were designed specifically for the purpose of this study. That means these instruments have not been tested in other studies yet. Interrater reliability assessments indicated a satisfactory level of agreement, but the results based on CTBS and programming quality rubric scheme deserve to be interpreted with caution. Some indicators of the computational thinking behavior are dependent on the environment used. For instance, the computational thinking component algorithmic design category of the scheme and encompasses all utterance and actions with the purpose of designing an algorithmic solution to a problem. The programming task in this study was designed in Scratch, for which the only way to create algorithmic solutions was to drag and drop code chunks together. If another programming environment were used, or indeed different programming problems, or even other cohorts of participants, other indicators may be identified. This potentially limits the generalization of the results of the study.

6 Conclusion and future work

Computational thinking is promoted as the literacy of the 21st century and is already implemented in various curricula all over the world. Some refer to computational thinking even as the foundation of programming and CS (Lu & Fletcher, 2009). Thus, the goal of this study was to analyses the role of computational thinking in promoting high quality programming products. Results showed that the answer to the question of how computational thinking is related to program quality depends on whether computational thinking is seen as a set of general conceptual understandings or a set of procedural skills in use. The results of our study found that computational thinking as general conceptual knowledge (such as that used to solve Bebras challenges) was not significantly related to program quality. On the other hand, we found that computational thinking as a set of procedural skills applied in practice was significantly related to programming quality, even when controlling for general intelligence. Thus, when discussing the role of computational thinking in developing computer programming capacity, we suggest that educators and policy makers focus on the importance

of cultivating computational thinking procedural capabilities rather than in more abstract, knowledge-based and context free forms.

There are several potential avenues for research to build upon the results of this study. Visual programming environments such as Scratch are usually used to introduce computational thinking or programming concepts to people who have no knowledge about programming, as was the case in this study. In future, researchers could analyse how computational thinking is applied when experienced programmers solve a programming task in a programming language such as Java or C++. The way programmers approach problems develops over time as they gain more knowledge (Teague & Lister, 2014). It is possible that the level of computational thinking for experienced programmers differs from the level of novices, which might mediate the relationship between both concepts. A range of different tasks could be examined, for instance, to gauge differences in computational thinking prevalence and relationships to program quality for tasks with more closed solutions as opposed to being more open-ended in nature. Future research could attempt to analyze all concepts individually rather than in pairs as an alternative way to examine the relationship between the constructs in question. In any case, it is intended that the frameworks and methods presented in this paper provide a strong foundation for these future analyses.

Acknowledgements None.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

Declarations

None.

Conflict of interest None.

References

Anderson, J. R. (2015). *Cognitive psychology and its implications* (8th.). New York, NY: Worth Publishers

Angeli, & Giannakos, M. (2020). Computational thinking education: Issues and challenges. *Computers in Human Behavior*, 105, 106185. https://doi.org/10.1016/j.chb.2019.106185

Araujo, A. L. S. O., Andrade, W. L., Guerrero, D. D. S., & Melo, M. R. A. (2019). How many abilities can we measure in computational thinking? A study on Bebras challenge. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 545–551). https://doi.org/10.1145/3287324#issue-downloads

Atmatzidou, & Demetriadis, S. (2016). Advancing students' computational thinking skills through educational robotics: A study on age and gender relevant differences. *Robotics and Autonomous Systems*, 75, 661–670. https://doi.org/10.1016/j.robot.2015.10.008

Banks, S. H., & Franzen, M. D. (2010). Concurrent validity of the TONI-3. *Journal of Psychoeducational Assessment*, 28(1), 70–79. https://doi.org/10.1177/0734282909336935

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, Vancouver, Canada*

Bower, M., & Hedberg, J. G. (2010). A quantitative multimodal discourse analysis of teaching and learning in a web-conferencing environment–the efficacy of student-centred learning designs. *Computers & Education*, 54(2), 462–478

Brown, L., Sherbeernou, R. J., & Johnson, S. K. (1997). *Test of nonverbal intelligence-3*. Austin, TX: PRO-ED

Bull, G., Garofalo, J., & Hguyen, N. R. (2020). Thinking about computational thinking. *Journal of Digital Learning in Teacher Education*, 36(1), 6–18. https://doi.org/10.1080/21532974.2019.1694381

Cansu, F. K., & Cansu, S. K. (2019). An overview of computational thinking. *International Journal of Computer Science Education in Schools*, 3(1), 17–30. https://doi.org/10.21585/ijcses.v3i1.53

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Hillsdale, N.J.: L. Erlbaum Associates

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2014). *Introduction to algorithms* (3rd ed.). Cambridge, MA, London: MIT Press

Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., & Woollard, J. (2015). *Computational thinking - A guide for teachers*. Swindon: Computing at School. http://eprints.soton.ac.uk/id/eprint/424545

Dagienė, V., & Futschek, G. (2008). Bebras international contest on informatics and computer literacy: Criteria for good tasks. In R. T. Mittermeir & M. M. Sysło (Eds.), *Informatics Education - Supporting Computational Thinking: Third International Conference on Informatics in Secondary Schools - Evolution and Perspectives, ISSEP 2008 Torun Poland, July 1-4, 2008 Proceedings* (pp. 19–30). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-69924-8_2

Dagienė, V., & Sentance, S. (2016). It's computational thinking! Bebras tasks in the curriculum. In A. Brodnik & F. Tort (Eds.), *Lecture Notes in Computer Science. Informatics in Schools: 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, Proceedings* (Vol. 9973, pp. 28–39). Cham: Springer Verlag. https://doi.org/10.1007/978-3-319-46747-4_3

Dagienė, V., & Stupuriene, G. (2016). Bebras - A sustainable community building model for the concept based learning of informatics and computational thinking. *Informatics in Education*, 15(1), 25–44. https://doi.org/10.15388/infedu.2016.02

Denner, J., Werner, L., Campe, S., & Ortiz, E. (2014). Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education*, 46(3), 277–296. https://doi.org/10.1080/15391523.2014.888272

Dolgopolovas, V., Jevsikova, T., Savulionienė, L., & Dagienė, V. (2015). On evaluation of computational thinking of software engineering novice students. In A. Brodnik & C. Lewin (Eds.), *IFIP TC3 Working Conference "A New Culture of Learning: Computing and next Generations"*. Vilnius, Lithuania: Vilnius University

Ezeamuzie, & Leung, J. S. C. (2021). Computational thinking through an empirical lens: A systematic review of literature. *Journal of Educational Computing Research*, Vol. 59, https://doi.org/10.1177/07356331211033158

Gadanidis, G. (2017). Five affordances of computational thinking to support elementary mathematics education. *Journal of Computers in Mathematics and Science Teaching*, 36(2), 143–151

Grover, S. (2011). Robotics and engineering for middle and high school students to develop computational thinking. In *Annual Meeting of the American Educational Research Association, New Orleans, LA*

Grover, S. (2017). Assessing algorithmic and computational thinking in K-12: Lessons from a Middle School Classroom. In P. J. Rich, & C. B. Hodges (Eds.), *Emerging Research, Practice, and Policy on Computational Thinking* (31 vol., pp. 269–288). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-52691-1_17

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, 25(2), 199–237. https://doi.org/10.1080/08993408.2015.1033142

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 86–93. https://doi.org/10.1145/1930464.1930480

International Society for Technology in Education [ISTE] & the Computer Science Teachers Association [CSTA] (2011). Operational definition of computational thinking for K–12 education. Retrieved from https://csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf

Jin, K. H., & Charpentier, M. (2020). Automatic programming assignment assessment beyond black-box testing. *Journal of Computing Sciences in Colleges*, 35(8), 116–125

Kafai, Y. B., De Liema, D., Fields, D. A., Lewandowski, G., & Lewis, C. (2019). Rethinking debugging as productive failure for CS Education. In S. Heckman & J. Zhang (Eds.), *Proceedings of the 50th ACM technical symposium on Computer Science Education*. New York, NY: ACM

Knobelsdorf, M., & Frede, C. (2016, August). Analyzing student practices in theory of computation in light ofdistributed cognition theory. *In Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 73–81).

Korucu, A. T., Gencturk, A. T., & Gundogdu, M. M. (2017). Examination of the computational thinking skills of students. *Journal of Learning and Teaching in Digital Age*, *2*(1), 11–19. Retrieved from https://eric.ed.gov/?id=ED572684

Landis, R., & Koch, G. G. (1977). The measurement of observer agreement for categorial data. *Biometrics*, 33, 159–174

Lockwood, J., & Mooney, A. (2018). Computational thinking in secondary education: Where does it fit? A systematic literary review. *International Journal of Computer Science Education in Schools*, 2(1), pp. 1–20. https://doi.org/10.21585/ijcses.v2i1.26

Lu, J. J., & Fletcher, G. H. L. (2009). Thinking about computational thinking. In S. Fitzgerald (Ed.), *Proceedings of the 40th ACM technical symposium on Computer science education*. New York, NY: ACM

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

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Pearson Prentice Hall

McNicholl, R. (2019). Computational thinking using Code.org. *Hello World*, 4, 36–37. https://issuu.com/raspberry314/docs/helloworld04

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: a web tool to automatically evaluate scratch projects. In J. Gal-Ezer, S. Sentance, & J. Vahrenhold (Eds.), *Proceedings of the Workshop in Primary and Secondary Computing Education, London, United Kingdom, November 09 - 11, 2015* (pp. 132–133). New York: ACM. https://doi.org/10.1145/2818314.2818338

Moreno-León, J., Román-González, M., Harteveld, C., & Robles, G. (2017). On the automatic assessment of computational thinking skills. In G. Mark, S. Fussell, C. Lampe, m. schraefel, J. P. Hourcade, C. Appert, & D. Wigdor (Eds.), *CHI'17: Extended abstracts: proceedings of the 2017 ACM SIGCHI Conference on Human Factors in Computing Systems : May 6-11, 2017, Denver, CO, USA* (pp. 2788–2795). New York, New York: The Association for Computing Machinery. https://doi.org/10.1145/3027063.3053216

Pieterse, V. (2013). Automated assessment of programming assignments. *Proceedings of the* 3rd *Computer Science Education Research Conference*, *13*, 4–5. https://doi.org/10.5555/2541917.2541921

Portelance, D. J., & Bers, M. U. (2015). Code and tell: Assessing young Children's learning of computational thinking using peer video interviews with ScratchJr. In M. U. Bers & G. L. Revelle (Eds.), *IDC '15: Proceedings of the 14th international conference on interaction design and children* (pp. 271–274). New York: ACM

Posner, M. I., & Keele, S. W. (1968). On the genesis of abstract ideas. *Journal of Experimental Psychology*, 77, 353–363. https://doi.org/10.1037/h0025953

Poulakis, E., & Politis, P. (2021). Computational Thinking Assessment: Literature Review. *Research on E-Learning and ICT in Education: Technological, Pedagogical and Instructional Perspectives*, 111–128

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., & Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60. https://doi.org/10.1145/1592761.1592779

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

Schank, R. C., & Abelson, R. P. (1977). *Scripts, plans, goals and understanding: An inquiry into human knowledge structures*. Hillsdale, NJ: L. Erlbaum Associates. Artificial intelligence series

Schulz, K., & Hobson, S. (2015). *Bebras Australia computational thinking challenge tasks and solutions 2014*. Brisbane, Australia: Digital Careers

Schulz, K., Hobson, S., & Zagami, J. (2016). *Bebras Austrlia computational thinking challenge - tasks and solution 2016*. Brisbane, Australia: Digital Careers

Shivhare, & Kumar, C. A. (2016). On the Cognitive process of abstraction. *Procedia Computer Science*, 89, 243–252. https://doi.org/10.1016/j.procs.2016.06.051

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

Sipser, M. (2013). *Introduction to the theory of computation* (3rd ed.). Boston: Cengage Learning

Sternberg, R. J. (2017). Human intelligence. *Encyclopaedia Britannica*. Retrieved from https://www.britannica.com/topic/human-intelligence-psychology/Development-of-intelligence#ref13354

Suters, L., & Suters, H. (2020). Coding for the core: Computational thinking and middle grades mathematics. *Contemporary Issues in Technology and Teacher Education (CITE Journal)*, *20*(3). Retrieved from https://citejournal.org/volume-20/issue-3-20/mathematics/coding-for-the-core-computational-thinking-and-middle-grades-mathematics/

Tang, K. Y., Chou, T. L., & Tsai, C. C. (2020). A content analysis of computational thinking research: An international publication trends and research typology. *Asia-Pacific Education Researcher*, 29(1), 9–19. https://doi.org/10.1007/s40299-019-00442-8

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. In J. Whalley (Ed.), *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*. Darlinghurst, Australia: Australian Computer Society, Inc

Thalheim, B. (2009). Abstraction. In L. Liu, & M. T. Özsu (Eds.), *Springer reference. Encyclopedia of database systems*, 1–3. New York, NY: Springer

Tsai, Liang, J. C., Lee, S. W. Y., & Hsu, C. Y. (2021). Structural validation for the developmental model of computational thinking. *Journal of Educational Computing Research*, Vol. 59, https://doi.org/10.1177/07356331211017794

Türker, P. M., & Pala, F. K. (2020). A Study on students' computational thinking skills and self-efficacy of block-based programming. *Journal on School Educational Technology*, *15*(3), 18–31 (14 Seiten). Retrieved from https://imanagerpublications.com/article/16669/

Vaníček, J. (2014). Bebras informatics contest: Criteria for good tasks revised. In Y. Gülbahar & E. Karataş (Eds.), *Informatics in Schools. Teaching and Learning Perspectives: 7th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2014, Istanbul, Turkey, September 22-25, 2014. Proceedings* (pp. 17–28). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-09958-3_3

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

Weinert, F. E. (1994). Lernen lernen und das eigene Lernen verstehen. [Learning how to learn and understanding the own learning]. In K. Reusser, & M. Reusser-Weyeneth (Eds.), *Verstehen. Psychologischer Prozess und didaktische Aufgabe [Understanding. Psychological processes and didactical tasks.]* (pp. 183–205). Bern: Huber

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

Wu, B., Hu, Y., Ruis, A. R., & Wang, M. (2019). Analysing computational thinking in collaborative programming: A quantitative ethnography approach. *Journal of Computer Assisted Learning*, 35(3), 421–434. https://doi.org/10.1111/jcal.12348

Zha, S., Jin, Y., Moore, P., & Gaston, J. (2020). Hopscotch into Coding: introducing pre-service teachers computational thinking. *TechTrends*, 64(1), 17–28. https://doi.org/10.1007/s11528-019-00423-0

Authors and Affiliations

Kay-Dennis Boom[1] · Matt Bower[2] · Jens Siemon[1] · Amaël Arguel[3]

✉ Matt Bower
matt.bower@mq.edu.au

[1] Department of Vocational and Business Education, University of Hamburg, Hamburg, Germany

[2] School of Education, Macquarie University, Building 29WW Room 238, Balaclava Rd North Ryde, 2109, NSWSydney, Australia

[3] CLLE, University of Toulouse, CNRS, Toulouse, France