Investigating challenges of debugging tasks in an undergraduate computational thinking course

Min Huang
College of Education
Texas Tech University
min.huang@ttu.edu

Jongpil Cheon
College of Education
Texas Tech University
Jongpil.Cheon@ttu.edu

Eunsung Park
College of Education
Tennessee Tech University
epark@tntech.edu

Xueni Fan
College of Education
Texas Tech University
rebekah.Fan@ttu.edu

Abstract: While debugging is a critical application of computational thinking (CT), it has not been getting enough attention in CT research studies in higher education. This exploratory qualitative study aims to identify students’ debugging challenges in Scratch, a block-based programming language. We analyzed students’ coding journals through open coding and thematic analysis. The findings show the most challenging block categories for the undergraduates are Control, Operators, and Variables and the top three debugging projects students had issues with were the projects using nested repeat, the multiple operators, and the nested if/else projects that relate to Control, Operator, and Variables blocks. Insufficient understanding of CT concepts and shortage of CT practice may be the main reasons for the challenges. We suggested some pedagogical strategies to support teaching and learning computational thinking skills through debugging activities.

Introduction

Computational thinking (CT) is widely acknowledged to be more than programming but a set of problem-solving skills (Shute et al., 2017; Wings, 2006), which is critical to the new generations of students to fully understand and participate in the computer-based world (Román-González et al., 2018). There has been considerable interest in incorporating CT into K-12 education (Barr & Stephenson, 2011). As a means of higher cognitive skill, CT calls for more attention from both researchers and practitioners in higher education too (Pérez-Marín et al., 2020). However, despite increasing attention to the field, there is still no consensus in regard to the definition and basic components of CT. Debugging is considered to be an important component of CT by many scholars (Brennan & Resnick, 2012; Cruz Castro et al., 2021; Jaipal-
Jamani & Angeli, 2017; Shute et al., 2017). Debugging process of finding and fixing errors deepens learners’ conceptual understanding and improves their problem-solving skills.

Previous research on debugging mainly focuses on how to improve debugging efficiency in practice. To help learners or programmers decrease bugs in programming and find bugs more effectively after programming, different approaches were adopted to classify and analyze the existing bugs (Mccauley et al., 2008). However, as most of the relevant research was based on text-based syntactic languages, they might not be adapted to other programming contexts such as block-based languages, which are widely used for teaching beginner learners. More importantly, although the analysis of bugs might disclose some underlying causes like a misunderstanding of concepts or cognitive limitations, they do not directly reflect their real thinking process or mindset. Therefore, this study utilized coding journals as data sources to study the natural cognitive debugging process of learners in an online undergraduate computational thinking course. We focused on the challenges of debugging most students have faced in Scratch programming and the reasons behind their struggle from the perspective of CT.

Literature Review

Construct of Computational Thinking (CT)

CT was first introduced by Seymour Papert, and he claimed that computer models could help children with intangible and abstract knowledge by giving them concrete form, and helping them develop metacognitive skills (Papert, 1980). But increasing attention has been given to it till CT was defined by Wing as a thinking process coming before computational technology (Wing, 2006). In 2012, Wing updated the CT definition, emphasized its essential role of abstraction and automation, and pointed out CT foundations in math and engineering. She clarified CT definition by stating that that CT was related to cognition rather than artifacts. The application of CT can support humans and machines to execute the solution effectively (Wing, 2012).

Although there is no agreement on whether the nature of CT is an approach, a skill, a thinking process, or a mindset, and whether it can be independent of using a computer, scholars agree that CT is not a single skill, but a combination including various complicated abilities (Cruz Castro et al., 2021; Jaipal-Jamani & Angeli, 2017; Lu et al., 2022; Peteranetz et al., 2018; Romero et al., 2017; Shute et al., 2017a; Weintrop et al., 2016; Wing, 2008, 2012). Among many components, abstraction (Brennan & Resnick, 2012.; Cruz Castro et al., 2021; Jaipal-Jamani & Angeli, 2017; Peteranetz et al., 2018; Shute et al., 2017; Wing, 2008, 2012), algorithm (Brennan & Resnick, 2012; Buitrago Flórez et al., 2017; Castro et al., 2021; Jaipal-Jamani & Angeli, 2017; Peteranetz et al., 2018; Shute et al., 2017), problem-solving skills (Buitrago Flórez et al., 2017; Shute et al., 2017a; Weintrop et al., 2016; Wing, 2012), debugging (Brennan & Resnick, 2012.; Cruz Castro et al., 2021; Jaipal-Jamani & Angeli, 2017; Shute et al., 2017), and generalization (Cruz Castro et al., 2021; Peteranetz et al., 2018; Romero et al., 2017; Shute et al., 2017) are common features.

Since Brennan and Resnick’s (2012) CT framework covers the most common feature mentioned above, their framework will be adopted in this paper as a conceptual framework. Their CT framework was developed from programming on the platform of Scratch, a block-based programming language, to allow their users to create and share their programming projects. Scratch is popular in programming for novice beginners, especially beginners in K 12. They defined CT as CT concepts, CT practices, and CT perspectives. CT concepts refer to concepts needed for programming, including sequences, loops, events, conditionals, operators,
and data. CT practice means specific practices during programming, including being incremental and iterative, testing and debugging, reusing and remixing, and abstracting and modularizing. CT perspective is about perspectives on the world around them and themselves, including expressing, connecting, and questioning.

**Debugging**

As an important part of CT, there is also no consensus on debugging, regarding its constructs. Debugging requires both knowledge and skills (Xie et al., 2019). Learners need to know some basic knowledge of programming constructs (Tew & Guzdial, 2010), such as syntactic rules of coding language to specify the structure or form of codes. Debugging is a practical skill with which basic programming principles and mathematics are applied to find problems, isolate sources, identify errors, and fix bugs that prevent the codes from executing designed functions properly (Liu et al., 2017). Debugging is also defined as a serial and iterative cognitive process to seek the reasons why desired results are not achieved in a program (Wong & Jiang, 2018). It’s highly demanding for some high-order cognitive skills such as logical thinking, and mathematical thinking.

The complexity of debugging makes its practice quite challenging even for professional programmers. Researchers have investigated debugging challenges from different perspectives. One of the approaches is to study the causes of bugs. Early research mainly attributed the bugs to the complexity of programming languages. Later, it was acknowledged that there are underlying reasons for the bugs (Bonar & Soloway, 1985): such as a misunderstanding or partial understanding of knowledge such as a misunderstanding or partial understanding of knowledge (Perkins and Martin, 1986; Pea, 1986). Ko and Myer (2005) broadened the boundary of the research from cognitive limitations to environmental factors, including programming systems and external environments. The research extends our understanding of the roots of the bugs but is not specific enough as sometimes bugs are rather contextualized. Bugs that frequently appeared in syntactic programming language might not appear in Block-based languages. For example, it was believed that many bugs were the result of a discrepancy between natural language and programming languages (Pea, 1986). Similarly, Spohrer and Soloway (1986) stated that one of the causes of bugs was “data-type inconsistency” which means there are different requirements or rules regarding different data types. In block-based languages, the probability of making these types of errors is very low. In another word, more attention could be cast on a “breakdown” between goals and plans (Spohrer, Soloway, & Pope, 1985) or “cognitive limitations” (Ko & Myer, 2005) behind the bugs, which is more relevant to the individual development of CT.

Another approach to studying debugging is to determine the categories of bugs (Mccauley et al., 2008). As for block-based language, Frädrich et al. (2020) tried to categorize the bugs in Scratch into three categories: syntax errors such as no conditions in infinite loops, general bugs such as using a variable without defining it, and Scratch-specific bugs like missing erase all. They further divided the categories into 25 patterns, which cover most bugs novice learners might frequently encounter. This type of research is beneficial for our understanding of the properties of bugs. However, the detailed and complicated classification might increase the cognitive load of both teachers and students in memorizing and understanding, and classifying those patterns, and therefore might be hard to be applied in real education.

property of codes and a discrepancy between plan and goal (Johnson, 1990). There are two main strategies used in the process: comprehension strategies and isolation strategies (Yoon & Garcia, 1995). The formal is to comprehend the discrepancy and the latter is to search for clues, assume possible bugs and test them. Xu and Rajlich (2004) described the debugging process using six levels of Bloom’s taxonomy of cognitive learning, from “knowledge” through “comprehension”, “application”, “analysis”, “synthesis”, and “evaluation”. Some recent research employed eye-tracking technology to observe students’ debugging processes (Lin et. al., 2015; Beelders, 2022). Results showed that high-performance students tended to organize the codes in chunks and review them in a more logical manner. In contrast, low-performance students focused on the details of syntactic features and tended to review line by line aimlessly.

Most previous research adopted observational and anecdotal methods to study bugs (Mccauley et al., 2008). They observed learners’ programming processes or did some static analysis of the bugs to categorize bugs or trace the reasons behind the bugs. Observational studies provide rich descriptive data with high accuracy, but it is rather time-consuming and difficult to free from researchers’ bias. More importantly, it’s not possible to know the real thinking and mindset of the subjects through observation. Studying debugging through bugs may have the same issue as the bugs may sometimes eliminate the complex cognitive process of debugging.

To supplement previous research, we employed coding journals with prompts as the main data source to identify the challenges novice adult learners frequently meet when debugging Scratch programs. The reflective coding journals gave students chances to record their struggles, dilemmas, trials, failures, and exploration (Phelps, 2005), which is aligned with the non-linear, learning-from-failure process of debugging. Moreover, the journals also help facilitate learning as it solidifies the connection between their experience and the meaning (Denton, 2011). Our purpose is neither to study the bugs or the causes of the bugs in a holistic way nor explore learners’ computer debugging skills. Instead, we focus on the challenges of debugging most students have met with in Scratch programming, and explore the reasons behind their struggle from the perspective of CT. The research questions are as follows:

- Which Scratch block categories are more difficult for students to debug?
- What debugging issues do students face and why?

**Methods**

**Data collection**

This study was conducted with 74 students who enrolled in an online course, “Computing and Information Technology” at a large public university in the southwest of the States. The course was one of the core curriculum courses for undergraduate students, and it was designed to use Scratch as a computational tool to improve students’ digital literacy, problem-solving, and critical thinking. The course was composed of 14 modules, and students are encouraged to create, share, and reflect on their own programming projects. The debugging practice in this course contained six debugging projects in which students were asked to fix bugs or errors in the given projects to implement the required functionalities of Scratch they had learned in previous modules. More specifically, eight-block categories (i.e., Control, Motion, Looks, Sound, Events, Control, Sensing, Operators, and Variables) had to be properly used. It was also required to use key CT concepts such as if-else, multiple if-else, repeat, repeat until, operator, and variables in
each project. After finishing their projects, students were asked to share their projects and reflect on their programming process in coding journals with pre-structured prompts by the instructor. We collected the coding journals (n = 74) in which students reflected on their debugging practices with given structured prompts:

- Among nine Scratch Block categories, which Block category is most difficult for you when completing the debugging projects?
- What issues did you face in the six debugging projects?

Data Analysis

The data was analyzed to identify the main themes using open coding and thematic analysis (Braun & Clarke, 2006; Clarke et al., 2015; Terry et al., 2017). First, the six debugging projects were analyzed to search for the main bugs. After that, students’ coding journals were read several times to ensure familiarization with the data. Then two main researchers of the study coded the data together to generate the themes. To achieve reliability and validity, they reviewed and discussed every theme together. When there is any disagreement, a third coder who is the designer and instructor of the course joined the discussion to help reach an agreement.

Results

The findings show that Control block category was the most difficult (n = 34) followed by Operators (n = 27) and Variables (n = 26) amongst the nine Scratch Block categories when debugging. In Scratch, Control block is used to control the events and movements of the sprites under certain circumstances. Operators provide support for mathematical, logical, and string (text) expressions (Brennan & Resnick, 2012), with which programmers can compare variables and values, do mathematical calculations, and work with strings. Variable is self-defined by the users to store, retrieve and update numerical values like speeds, and scores. The top three debugging projects students had issues with were the projects using nested repeat (n = 33), the multiple operators (n = 23), and the nested if/else (n = 22) projects that relate to Control, Operator, and Variables blocks as well.

Regarding the Control blocks, the challenges are caused by the difficulty of logical thinking, the nature of complexity, and the lack of understanding of each block. Since conditional blocks are in this category, students were confused about the order of actions. A student stated that “It was hard to know what order blocks were supposed to go,” and the other student said, “order matters more than anything. 15 students mentioned that control blocks require a lot of logical thinking. Next, 12 students stated that the nature of the combination with other blocks caused the complexity of using control blocks. More specifically, when control blocks are combined or nested with operators and variables, it is difficult to decompose them into smaller actions when testing larger blocks. Lastly, we found that some students did not fully understand what each control block functions. They said that “it is confusing to understand which control blocks to use and when to use” and “The repeat and repeat forever as well as the if/else blocks were a little confusing.” The low understanding level could be called fragile knowledge (Perkins & Martin, 1986).

Since operators have numerous combination options with other blocks and require fundamental mathematical thinking, it would be hard to track pre-defined variables. Thus, the two categories are challenging for students. First, many students believed operator blocks were difficult because they manage various conditions and need to be nested into other blocks. One
student stated that “It allows many possibilities and options, therefore make debugging more
difficult,” and another student said that “The operators block I believe there are so many options
I get a bit overwhelmed.” Second, multiple or nested operator blocks could be difficult for
students with poor math backgrounds to find and resolve errors. Since the operators are not
simple math but are related to complicated control over sprites or actions, learners need more
practices to utilize operators (Zhang & Nouri, 2019). Third, variables are self-defined, so
students should be aware of how variables were assigned and where the variables were used.
Two statements from the coding journal are as follows: “I didn’t know how variables were set up
and just confused”; “It’s hard to figure out what the purpose it or how it should be used.” The
last finding is the lack of debugging strategies. Students underestimated the debugging tasks, and
they did not know efficient ways to resolve issues. Although the debugging process should be
taken more seriously even when creating an actual programming project, it seems like they
oversimplify the task in the six debugging projects. For instance, they described their individual
debugging process as “move around the blocks to notate what changes and what does and, once
again,” “just try and try differently,” or “I kept clicking the green flag.” A few students used the
decomposition strategy to break down a project into separate blocks to test each block. Some
students used forward or backward debugging strategies to test blocks step by step.

Discussion

Few empirical studies on CT have shed light on how to understand learners’ debugging
challenges in Scratch. This study investigated students’ debugging practices from a qualitative
approach. Our findings show that the three block categories including Control operator and
Variables were more challenging and were possible causes of the obstacles even after completing
a number of programming projects.

There are many reasons behind students’ Scratch debugging challenges. Similar to
Perkins and Martin’s (1986) “fragile knowledge” and Ko and Myer’s “knowledge breakdown”,
our findings also suggested short of knowledge could explain learners’ debugging failure.
According to Perkin and Martin, fragile knowledge could be classified into missing knowledge (I
don’t know), inert knowledge (I don’t remember), misplaced knowledge (It’s not applicable to
the current context), and conglomerated knowledge (Two structures were combined incorrectly)
(Perkin & Martin, 1986). In our study, students attributed their difficulties in debugging to their
missing knowledge about some blocks and conglomerated knowledge when using variables and
operators simultaneously. Their reflection proves the importance of knowledge construct in
debugging.

More importantly, we found the knowledge gap of the learners originated from some CT
conceptual breakdown. In the nine blocks of Scratch, most of them only involve one or a few CT
concepts. For example, Motion block mainly relates to the concept of sequence: a sequence of
programming instructions finally produces a certain action, which is comparatively easy,
especially for debugging. However, Control blocks involve multiple concepts of CT. To
correctly debug, one has to be familiar with at least 4 concepts: conditional, event, sequences,
and loops, which increase the difficulty of understanding. Operator and variable blocks most of
the time are combined to achieve some functions. To debug, one has to understand the concepts
of operator and data. Operators enable coders to manipulate both numerical data and textual data,
and Variables deal with data storing, retrieving, and updating. The conceptual complexity of
those blocks brings challenges to novice programmers when they are debugging.
Conceptual weakness is not the only reason for the debugging challenges, another reason is related to their CT practices. As stated before, debugging is also a skill. It stands on the end of productive skills on the continuum from reproductive skills to productive skills (Romiszowski, 1993). According to Romiszowski (1993), practice is essential to skill development. It bridges “knowing the relevant CT concepts” and “learning how to make use of the concepts to debug”. However, debugging practices, as a combined process of constant testing and code correction (Xu & Rajlich, 2004), are iterative and incremental (Brennan & Resnick, 2012). In another word, they are quite labor-intensive and difficult (Araki, at. al., 1991). However, through the journals, we found many students underestimated the difficulty of debugging, which sometimes lead to their debugging failure. Besides, codes containing mixed use of Control, Operator, and Variables blocks sometimes need to be modularized and decomposed to help emerge the errors. Novice programmers short of such practices may find debugging very frustrating. Our findings also emphasized the importance of strategies in debugging practice, which is consistent with Brennan and Resnick’s research (2012). Based on Brennan and Resnick, it’s essential to develop strategies for both “dealing with” and “anticipating” problems in codes. For novice programmers, it’s hard to develop those strategies by themselves. According to their coding journals, most students continuously used the “trial and error” strategy and gave up when the strategy failed.

Students’ challenges with debugging reflected their insufficient understanding of CT concepts and shortage of CT practices. Our finding implies that pedagogical strategies should pay special attention to providing conceptual knowledge in programming classes because knowledge construction is fundamental for learners to develop skills and cognitive abilities. One way to facilitate students' conceptual learning is to build their knowledge through practice. We suggest that compared to complicated creative projects, designing small and specific tasks involving one or a few concepts is more effective, especially for novice learners. We also suggest providing scaffoldings including explicit instructions or resources for those comparatively difficult block categories. For example, when teaching adult students Variables, it would be beneficial to directly explain the differences between Variables as a programming block in Scratch and variables as mathematical knowledge in their minds.

Debugging is never a linear practice; therefore it supports students by learning from failure (Kafai et al., 2019; Michaeli & Romeike, 2019). As teachers, it is important to help students have rational anticipations of the potential difficulties of debugging and get them mentally prepared for the complicated debugging process. Sometimes, demonstrating some complex practices can be beneficial to students, such as showing them how to filter irrelevant data to concentrate on the main logic first or how to subdivide a program into smaller chunks to find the errors more efficiently. Providing debugging procedures or strategies to guide students, such as systematic forward/backward debugging, decomposing debugging, or peer debugging, are key components of computational thinking practice (Brennan & Resnick, 2012). We may consider inductive reasoning to teach programming to enhance computational thinking and encourage interaction among students.

Our study is limited to lacking the triangulation of the transcription of students’ journals, and systematic and comprehensive analyses of the causes of students’ difficulties. Future studies might include a wider range of students and add other data resources like surveys, interviews, and their performance in the course because the current study is restricted to coding journals only.
References:


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](https://doi.org/10.1145/1929887.1929905)


Kafai, Y., Biswas, G., Hutchins, N., Snyder, C., Brennan, K., Haduong, P., ... & Danish, J. (2020). Turning bugs into learning opportunities: understanding debugging processes, perspectives, and pedagogies. *Proceedings of the 14th international conference of the learning science (ICLS).* [https://repository.isls.org/bitstream/1/6661/1/374-381.pdf](https://repository.isls.org/bitstream/1/6661/1/374-381.pdf)


Lowe, T. (2019, October). Debugging: The key to unlocking the mind of a novice programmer?. In 2019 IEEE Frontiers in Education Conference (FIE) (pp. 1-9). IEEE.


Wing, J. M. (2012, October 26) Computational thinking [Keynote]. Microsoft Asia Faculty Summit, Tianjin, China.


