Developing Computational Thinking Skills in Elementary Students

Sarah E. Van Loo

Michigan State University

Developing Computational Thinking Skills in Elementary Students

As a science, technology, engineering, arts, and math (STEAM) educator, one of the subjects I teach is coding. My elementary school students enjoy coding; however, some students have a difficult time with large, complex coding projects because they struggle with breaking problems into smaller problems and also with debugging their code when it does not work properly. This is an issue for every age level from elementary to university students. However, because I teach elementary school, that age level draws my attention the most.

To prepare for the coming school year, I want to learn specific strategies to help students develop computational thinking and coding skills, including breaking down big problems into smaller parts and debugging their code. In an effort to learn these strategies, I conducted a literature review of peer-reviewed research articles that address the issues of computational thinking and computer science instruction in K-12, and where possible, in K-5. Some articles focused on computer science as a subject to teach computational thinking (Angeli et al., 2016), while some investigated the possibility of teaching computational thinking through other subjects, including mathematics, biology, language, and programming (Hsu, Chang, & Hung, 2018).

The concept of computational thinking was first introduced by Papert (1980), when he suggested that children may be able to construct their knowledge better through working with a physical object, an "object-to-think-with" (p. 11). Papert's LOGO Turtle, for example, is a robot-like physical object that children can write programmed instructions for. By programming and testing the instructions on the physical object, children think with the object and construct their knowledge. Wing (2006) pushed the idea of computational thinking further when she

emphasized that it was not just a skill for computer scientists and programmers, but was actually a "fundamental skill for everyone" (p. 33). In the school district where I teach, we are approaching computational thinking as a fundamental skill for everyone as we continue the ongoing process of rolling out computer science curriculum district wide. In every building, students begin learning computer science by first grade at the latest.

There is debate over what exactly computational thinking is. However, common components or skills researchers agree upon include decomposition, abstraction, algorithm, debugging, iteration, and generalization (Shute, Sun, & Asbell-Clarke, 2017). The two specific components of computational thinking I identified in my problem statement are decomposition, the skill of breaking a big problem into smaller ones (Angeli et al., 2016) and debugging, the ability to find errors in code and fix them (Shute et al., 2017). However, upon beginning my research, I saw the importance of all of the component skills of computational thinking and expanded my problem statement to include the teaching of all six components.

Israel, Wherfel, Pearson, Shehab, & Tapia (2015) identified some specific interventions for elementary students struggling with computational thinking skills, such as using universal design for learning (UDL), utilizing explicit instruction, and encouraging student-to-student collaboration. Vihavainen, Airaksinen, & Watson (2014) identified pair programming, real-world problem solving, and game maker platforms as successful interventions at the university level. Although Vihavainen et al.'s recommendations were based on research with university-aged students, they may also be successful with elementary-aged students. Overall recommendations for teaching computational thinking include implementing a K-6 computer science curriculum (Angeli et al., 2016) that is integrated into other content areas, especially

math and science (Israel, Wherfel, et al., 2015). Computer science should include a scaffolded approach to instruction, where students transition from one programming language or coding platform to increasingly more challenging ones as it is developmentally appropriate to do so (Touretzky Marghitu, Ludi, Bernstein, & Ni, 2013).

Literature Review

To address my problem of teaching computational thinking skills to support computer science skills, I read 18 peer-reviewed articles dating between 2006 and 2018. I searched for articles about teaching computational thinking skills through computer science curriculum and interventions for computational thinking skills. Most research on computational thinking and coding was focused at the university level. There was little focused on K-12, and even less focused on K-5, most likely because computer science is still not a regular part of many elementary schools' curricula (Mladenović, Boljat, & Žanko, 2017). According to Hsu et al. (2018) computational thinking can be taught through various subjects including math, biology, computer science, and languages. However, because my research problem is centered around developing computational thinking skills related to computer science, I sought out research that highlighted computer science as a subject for teaching computational thinking.

Computational Thinking Defined

There is not one single, agreed-upon definition of computational thinking. According to Kong (2016), this can make it difficult to define it in the context of K-12 education. However, Shute et al. (2017) identified six common components or skills researchers agree upon. Decomposition is defined as "breaking complex problems into simpler ones" (Angeli et al., 2016, p. 49). Abstraction is simplifying problems to their component parts to make them easier

to think about and understand (Flórez, 2017). Algorithm is a "sequence of steps or other problem-solving operations" (Dwyer, Hill, Carpenter, Harlow, & Franklin, 2014, p. 514). Debugging is defined as "identifying and fixing errors when solutions do not work as expected" (Shute et al., 2017, p. 145). Iteration means repeating the process until the "ideal result" is achieved (Shute et al., 2017, p. 153). Finally, generalization refers to transferring computational thinking skills to other situations to aid in solving problems (Shute et al., 2017).

The Need to Teach Computer Science in Elementary Settings

As digital technology grows increasingly important and present in our daily lives, "it is imperative individuals have the education, knowledge, and skills to critically understand the technological systems they use, as well as to be able to troubleshoot and problem solve when things go wrong" (Angeli et al., 2016, p. 47). Not only are technology skills important for computer scientists, Israel, Wherfel, et al. (2015) argued, but computing skills are necessary for other professions like journalism and creative arts. That said, there is a specific need for teaching computer science in elementary education so that children can begin to develop these skills as early as possible. According to Angeli et al. (2016), when students are exposed to computer science instruction in elementary school, they develop computational thinking skills, they become content producers and not just consumers of content, and they think about other disciplines differently. Other benefits for students include building higher-order thinking skills and collaborative problem-solving skills and fostering positive attitudes about computer science and its related skills (Israel, Wherfel, et al., 2015).

Challenges of Teaching Computational Thinking in Elementary Settings

Despite the argument that computer science skills are critical, when it comes to teaching computational thinking at the elementary level, several challenges emerge. Angeli et al. (2016) identified these as a possible ability gap between what students are able to do and what they are expected to do, the question of what content to teach, and the question of what teachers need to know to be able to teach computer science. Israel, Pearson, Tapia, Wherfel, & Reese (2015) also identified challenges for special populations like students with disabilities and those at risk for academic failure due to poverty.

Possible ability gap due to age. Some research suggests an ability gap that is created because of the need to introduce computational thinking before high school (Angeli et al., 2016) and the perceived challenge for younger students to be able to think abstractly (Piaget, 1962), a component skill of computational thinking (Mladenović et al., 2017). According to Piaget (1962), abstraction is a skill that children will acquire after age 11 when they enter the formal operations stage, so elementary school students are too young to be able to think abstractly. Yet more recent research suggests that high school is too late to introduce computer science for the first time due to its complex nature and relative importance (Angeli et al., 2016). Fortunately, there is evidence to show that, when "concrete reference systems are used to situate their thinking" (Angeli et al., 2016, p. 46), young children are able to think abstractly (Gibson, 2012). Through a series of sessions conducted with student ranging in age from five to six years old up to 14-15 years old, Gibson (2012) demonstrated that children can work with abstractions from a young age. Sessions began with students drawing two-dimensional shapes, then moved on to creating three-dimensional structures. First, a researcher described a two-dimensional image and

students drew the image based on the description, such as, “two circles connected by lines” (Gibson, 2012, p.36). As a group, students examined the drawings and determined and discussed what the criteria would be for two drawings to be the same. Sessions were conducted in an open-ended manner in such a way that students created and agreed upon the rules for assessing the drawings (Gibson, 2012). Additional sessions involved increasingly complex problems, such as identifying the shortest path between two points (Gibson, 2012). Young children are capable of abstract reasoning, like making a drawing based on oral instructions or learning computer science, Gibson (2012) argued, if they have a concrete setting to anchor it.

Possible ability gap due to poverty or disability status. Because of the nature of computer science classes, basic computer navigation skills like using a mouse, dragging, and double-clicking are important. Yet they can be a challenge for students in poverty who may lack computer experience due to limited access to technology (Israel, Pearson, et al., 2015). Also, students with disabilities may struggle with accessing the computer science curriculum, although accommodations described later helped students with disabilities, except in some cases of severe disabilities (Israel, Pearson, et al., 2015). A computing environment, Israel, Pearson, et al. (2015) argued, was discovered to be one where struggling learners thrived. “Even the students who are most challenged find computers friendly because it doesn't care if you're right or wrong” (Israel, Pearson, et al., 2015, p. 273).

What content to teach. Angeli et al. (2016) point out that just a few of the popular computer science programs in the United States include *Beauty and Joy of Computing*, *Code.org, Computer Science Principles*, *Exploring Computer Science*, and *Project Lead the Way (PLTW)* (p. 48). Each of these programs has a slightly different approach and different content, presenting

schools with a big decision when it comes to choosing a program. Schools could opt not to use packaged curriculum and build their own, resulting in another series of choices for schools to make.

Educator's knowledge. In addition to identifying what computer science content to teach, Angeli et al. (2016) also identified a question of what teachers need to know to be able to teach computer science, which is an extensive body of knowledge. Teachers need technological pedagogical content knowledge (TPCK) (Koehler & Mishra, 2008) including content knowledge about computational thinking and its component parts: decomposition, abstraction, algorithm, debugging, iteration, and generalization (Angelie et al., 2016). Teachers also require pedagogical knowledge about learners' difficulties in computational thinking and its component parts, general pedagogical knowledge about teaching students of a given age, and technological knowledge about how to use available technologies for teaching computational thinking (Angeli et al., 2016). Gibson (2012) also stated that it is critical for teachers to understand computer science if they are to teach it well, and argued that the best way to prepare teachers of computer science is for them to actually teach it. This can be done, Gibson (2012) stated, by introducing experimental computer science sessions where teachers and students are exposed to the new material together. Gibson (2012) argued that a teacher can be inexperienced as long as they are enthusiastic.

Specific Interventions for Computer Science Instruction

In my research, I sought out specific interventions for helping students who are struggling with coding. Although more research seemed to center around developing a general framework for computer science curriculum (Angeli et al., 2016), I did find some specific interventions for

computer science instruction. Specifically aimed at exceptional students like those with disabilities, the interventions posed by Israel, Wherfel, et al. (2015) could also be used to support struggling general education students. I also found specific interventions based on research at the university level (Vihavainen, Airaksinen, & Watson, 2014) that could be adapted to the elementary level.

Specific interventions aimed at the elementary level. According to Israel, Wherfel, et al. (2015) specific interventions for elementary students struggling with learning computer science skills include universal design for learning (UDL), explicit instruction, and student-to-student collaboration. The use of unplugged activities to teach computer science concepts can also be engaging and motivating for young students (Touretsky et al., 2013).

Universal design for learning (UDL). Universal design for learning (UDL) is a framework for "proactively addressing barriers to learning" (Israel, Wherfel, et al., 2015, p. 46). Through the application of UDL, teachers can support the needs of individual students by providing multiple means of representation, multiple means of action and expression, and multiple ways to engage students (Meyer, Rose, & Gordon, 2014). When teachers provide students with different methods for accessing materials, they are providing multiple means of representation (Meyer et al., 2014). In the context of teaching coding, this could include modeling how to use the coding software, sharing teacher-created code with students, or letting students view additional how-to tutorial videos (Israel, Wherfel, et al., 2015). Teachers can provide multiple means of action and expression when they are flexible about what students create (Meyer et al., 2014). As Israel, Wherfel, et al. (2015) noted, computer science instruction is inherently flexible, and teachers provide multiple means of action and expression when they

allow students to use templates, expand on a program someone else created, replicate a project someone else made, or create their own projects. When teachers encourage collaboration or provide choices in projects that utilize the same skills in different ways, they are using multiple ways to engage students (Meyer et al., 2014). Examples in computer science class can include student choice between structured, leveled lessons like in Code.org and open-ended project creation like in Scratch (Israel, Wherfel, et al., 2015).

Explicit instruction. For students who struggle with following multi-step directions, the use of explicit instruction can be invaluable (Israel, Wherfel, et al., 2015). Among other elements, Israel, Wherfel, et al. (2015) stated that explicit instruction can include beginning with clear goals, providing and modeling step-by-step instructions, using clear language and clarifying new terminology, providing for lots of guided practice, monitoring student performance, and providing immediate feedback. While explicit instruction should be carefully balanced with open-ended inquiry, it can go a long way toward reducing student frustration and stress (Israel, Wherfel, et al., 2015).

Student-to-student collaboration. Collaboration can easily be fostered in a computer science context, because many coding problems are open-ended and emphasize creativity (Israel, Wherfel, et al., 2015). Collaboration can take the form of cooperative learning where students work together to construct new knowledge or it can involve student-to-student help-seeking, where students look to their peers for help before asking a teacher (Israel, Wherfel, et al., 2015). Student collaboration can also form organically, as students naturally problem solve with their peers (Israel, Pearson, et al., 2015). Either way, students need to be taught the necessary skills to work in a collaborative environment, including how to ask a peer for help and how to

appropriately offer support to a peer (Israel, Wherfel, et al., 2015). The benefits can be great, though, as students gain new skills and higher self-efficacy for learning, the belief that they capable of being in control of their own learning and able to learn new things (Schunck, 2012). Further, students who typically struggle in other environments but are successful with the computer "take leadership roles in helping other students" (Israel, Pearson, et al., 2015, p. 273).

Use of unplugged activities. Unplugged activities are those that help students apply computer science concepts without the use of computers (Touretsky et al., 2013), usually in the form of games. Students "use their bodies or simple props to simulate algorithms illustrating important computer science concepts" (Touretsky et al., 2013, p. 611). Examples may include using a special deck of cards to create an algorithm, a set of step-by-step instructions for another student, or running a relay race where students take turns writing and debugging each other's code, checking it for errors. These are typically done in collaborative groups and may involve being up from their seats. These types of activities are kinesthetic and have been proven to help engage students with disabilities (Tourestsky et al., 2013).

Specific interventions aimed at the university level that could be adapted for or implemented at the elementary level. In addition to the interventions identified by Israel, Wherfel, et al. (2015), Vihavainen et al. (2014) identified many other interventions for computer science instruction as successful interventions at the university level. Although these recommendations for interventions resulted from research with university-aged students, pair programming, real-world problem solving, the use of game maker platforms and the use of concept maps may also be successful with elementary-aged students (Vihavainen et al., 2014).

Pair programming. One specific type of collaborative activity identified by Vihavainen et al. (2014) is pair programming, in which two students work together on one coding project (Flórez et al., 2017). They take turns in specific roles as they share a single computer to code, debug, and test their project. When students pair program, they develop communication skills, and they share the responsibility for inputting code and for finding errors in the code.

When my fourth graders code a game together, one student is in the role of driver and is responsible for holding the iPad and inputting code. The other student is in the role of navigator and is responsible for watching the driver, giving directions, and helping to catch mistakes. Just like driving a car, it can have negative consequences if the navigator tries to take over driving, so students learn to take turns. This is a strategy used by some professional computer scientists, and I teach students that they are learning and applying skills that professionals also use.

Real-world problem solving. Problem-based learning is when students' learning is undertaken in order to solve a problem (Hsu et al., 2018). When learning is situated in the context of a real-world problem, students are more curious, motivated, and favorable toward school (Angeli et al., 2016). This type of learning also draws from other subject areas. It "demands a wider range of content, simply because authentic real-world problems are usually multidisciplinary in nature" (Angeli et al., 2016, p. 52).

For example, I begin a first grade coding unit by introducing a problem where fictional characters want to write their own story and make an animated movie about it. Students are excited to solve the problem by writing a story, an English Language Arts connection, and animating it on their own using Scratch Jr. to code the animation. Everything they learn about coding in Scratch Jr. is done in the context of preparing to animate their own story.

Game maker platforms. As Angeli et al. (2016) stated, it is important for students to become digital content creators. One engaging way to foster this is through game maker platforms (Flórez, 2017). In my third-grade computer science classes, students create their own games using the iPad app Hopscotch. During this unit, they learn computer science terminology in context, and they are excited and motivated as they create multiple games of increasing complexity. Students share their games with each other, referencing others' work to improve their own games. They ask each other questions about how game elements work, then go home and get their parents to play their games.

Concept maps. Concept maps are a tool that allows learners to make a graphical representation of knowledge and helps them organize the information, analyze it, and connect it to apply it to other subjects (Flórez et al., 2017). When first graders prepare to animate their own story using the Scratch Jr. coding environment, they use a graphic organizer to draw a storyboard and to plan other important information including the characters and the setting.

Recommendations for Teaching Computational Thinking

Overall recommendations for teaching computational thinking include implementing a K-6 computer science curriculum (Angeli et al., 2016) that is integrated into content areas, especially math and science (Israel, Wherfel, et al., 2015). Computer science should include a scaffolded approach to instruction (Touretzky, Marghitu, Ludi, Bernstein, & Ni, 2013).

Guided by students' interests, capabilities, and learning habits. According to Kong (2016), student "interest is the ideal motivation" (p. 378), and students' learning interests, their capabilities in creation, and their learning habits should provide a "theoretical ground" (p. 379) for any comprehensive computer science curriculum.

By immersing students in interest-driven learning at a level where they are capable of creating, they develop lifelong habits of creation, what Kong (2016) referred to as "the habit loop" (p. 379). First, learners imitate others' computer science projects to assimilate and accommodate computational thinking knowledge (Kong, 2016). Next, they analyze their own knowledge and decide what to use, combining it in an effort to create something new (Kong, 2016). Finally, in the last phase of the habit loop, staging, learners share their computational thinking skills and knowledge with others in authentic ways Kong (2016). Among many possible examples, this could include the sharing of Hopscotch games by third graders with their classmates or a robotics competition where students compete with their programmed robots to perform challenges. Kong (2016) argued that authentic expression could include "entertaining, engaging, equipping, or educating others" (p. 385).

Integrated into content areas or taught in isolation. Depending on the amount of time teachers have available, computer science can be taught in isolation, or integrated into other subjects, like math and science (Israel, Wherfel, et al., 2015). When taught in isolation, computer science instruction can be implemented through an approach that involves discrete lessons, like Code.org, or it can be implemented through open exploration where teachers conduct lessons through programming software and students create projects using the same software (Israel, Wherfel, et al., 2015). In cases where teachers do not have enough time to conduct separate coding lessons, they may integrate computer science into other content areas, such as math and science (Israel, Wherfel, et al., 2015). For example, when my fourth graders are coding games in Tynker, I teach them about the x-axis and y-axis, and they use that information to code their

game characters to move properly. Students are motivated to learn new concepts in the context of creating projects.

Based on a scaffolded approach to instruction. Computer science should be based on a scaffolded approach to instruction, where students transition from one programming language or coding platform to increasingly more challenging ones as it is developmentally appropriate to do so (Touretzky Marghitu, Ludi, Bernstein, & Ni, 2013). For example, students at a five-day camp progressed from learning Microsoft Kodu to Alice to Lego Mindstorms NXT-G with the result that concepts and skills learned in Kodu would transfer and be built upon as students continued learning in Alice and then NXT-G (Touretzky et al., 2013).

Includes specific lessons in certain computational thinking skills. Dwyer et al. (2014) found that students had difficulty improving their ability to write step-by-step instructions, related to algorithmic thinking, even after analyzing their own and others' first attempts at instruction writing. Therefore, Dwyer et al. (2014) argued, computer science curriculum should include lessons on procedural writing, including the importance of sequencing, putting steps in the correct order, and discussion about computer science vocabulary words. Students should also be given enough time to develop and test these skills as they learn them (Dwyer et al., 2014).

Includes computational thinking skills as a learning objective. Rather than only teaching students how to write code, Flórez et al. (2017) argued, computer science courses should explicitly include the development of component computational thinking skills as a learning objective. Although the knowledge of how to code is specific only to the platform it is used on, the component skills of computational thinking - decomposition, abstraction, algorithm, debugging, iteration, and generalization (Shute et al., 2017) can be learned and applied across

platforms and coding environments. This helps learners become proficient in programming and "may further develop their thinking abilities in a reasonable time" (Flórez et al., 2017, p. 837). Angeli et al. (2016) expanded on this, saying that the components of computational thinking should be taught holistically. Whereas some educators teach the component parts of computational thinking one at a time using a compartmentalized approach, they should all be taught together, "focusing on whole complex and authentic learning tasks, without losing sight of the individual elements that make up the complex whole" (Angeli at al., 2016, p. 52).

Conclusion

Computational thinking is a "fundamental skill for everyone" (Wing, 2006, p. 33), not just for computer scientists but for artists, journalists, and others (Israel, Wherfel, et al., 2015). It includes the component skills of decomposition, abstraction, algorithm, debugging, iteration, and generalization (Shute et al., 2017). One of the most common ways to develop computational thinking is through computer science instruction (Angeli et al., 2016), which is more extensive than just coding instruction (Flórez et al., 2017).

My students enjoy coding. However, there are times they get frustrated, especially when they struggle with breaking complex coding problems into smaller problems and with debugging their code when it does not work properly. I had hoped to find specific interventions to help with coding skills and to develop computational thinking. What I found, however, was much more interesting.

Instead of information about how to solve specific, individual problems with students as they arise in the classroom, what I found was the recommendation to implement a comprehensive computer science curriculum and how to do it effectively. When students learn

computer science, not just coding, they learn more than how to program in one language; they learn computational thinking skills that can transcend platforms and coding environments (Flórez et al., 2017).

Elementary schools should implement a comprehensive computer science curriculum (Angeli et al., 2016) that is integrated into content areas, especially math and science (Israel, Wherfel, et al., 2015). Computer science should include a scaffolded approach to instruction (Touretzky, Marghitu, Ludi, Bernstein, & Ni, 2013). It should be guided by students' interests, capabilities, and learning habits (Kong, 2016). It should include specific lessons in certain computational thinking skills (Dwyer et al., 2014), and it should include computational thinking skills as learning objects (Flórez et al., 2017).

In the context of a comprehensive computer science curriculum, however, educators can implement supports to aid struggling students. According to (Israel, Wherfel, et al., 2015), UDL, explicit instruction, and peer-to-peer collaboration can all support struggling learners. Setting computer science projects in the context of real-world problems can motivate students (Angeli et al., 2016). And utilizing concept maps to help students organize, analyze, and connect their knowledge to other subjects can support students as they learn computer science concepts (Flórez et al., 2017).

References

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 computational thinking curriculum framework: Implications for teacher knowledge. *Journal of Educational Technology & Society, 19*(3), 47-57. Retrieved from http://www.jstor.org.proxy1.cl.msu.edu/stable/jeductechsoci.19.3.47

Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students pre-instructional ability to develop algorithms and step-by-step instructions. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education - SIGCSE 14,* 511-516. doi:10.1145/2538862.2538905

Flórez, F. B., Casallas, R., Hernández, M., Reyes, A., Restrepo, S., & Danies, G. (2017). Changing a generation's way of thinking: Teaching computational thinking through programming. *Review of Educational Research, 87*(4), 834-860. doi:10.3102/0034654317710096

Gibson, J. P. (2012). Teaching graph algorithms to children of all ages. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education - ITiCSE 12*. doi:10.1145/2325296.2325308

Hsu, T., Chang, S., & Hung, Y. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education, 126*, 296-310. doi:10.1016/j.compedu.2018.07.004

Israel, M., Pearson, J. N., Tapia, T., Wherfel, Q. M., & Reese, G. (2015). Supporting all learners in school-wide computational thinking: A cross-case qualitative analysis. *Computers & Education, 82*, 263-279. doi:10.1016/j.compedu.2014.11.022

Israel, M., Wherfel, Q. M., Pearson, J., Shehab, S., & Tapia, T. (2015). Empowering K–12 students with disabilities to learn computational thinking and computer programming. *TEACHING Exceptional Children, 48*(1), 45-53. doi:10.1177/0040059915594790

Koehler, M. J., & Mishra P. (2008). Introducing TPCK. In AACTE Committee on Innovation and Technology (Eds.), *Handbook of Technological Pedagogical Content Knowledge (TPCK) for educators* (pp. 3–29). New York, NY: Routledge. Retrieved from http://www.punyamishra.com/wp-content/uploads/2008/05/koehler_mishra_08.pdf

Kong, S. (2016). A framework of curriculum design for computational thinking development in K-12 education. *Journal of Computers in Education, 3*(4), 377-394. doi:10.1007/s40692-016-0076-z

Meyer, A., Rose, D. H., & Gordon, D. (2014). *Universal design for learning: Theory and practice*. Wakefield, MO: CAST. Retrieved from http://udltheorypractice.cast.org/home?1

Mladenović, M., Boljat, I., & Žanko, Ž. (2017). Comparing loops misconceptions in block-based and text-based programming languages at the K-12 level. *Education and Information Technologies, 23*(4), 1483-1500. doi:10.1007/s10639-017-9673-3

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Piaget, J. (1962). The stages of the intellectual development of the child. *Bulletin of the Menninger Clinic, 26*(3), 120.

Schunck, D. H. (2012). *Learning theories: An educational perspective* (6th ed.). Boston, MA: Pearson.

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

Touretzky, D. S., Marghitu, D., Ludi, S., Bernstein, D., & Ni, L. (2013). Accelerating K-12 computational thinking using scaffolding, staging, and abstraction. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE 13,* 609-614. doi:10.1145/2445196.2445374

Vihavainen, A., Airaksinen, J., & Watson, C. (2014). A systematic review of approaches for teaching introductory programming and their influence on success. *Proceedings of the Tenth Annual Conference on International Computing Education Research - ICER 14,* 19-26. doi:10.1145/2632320.2632349

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