CHAPTER 2

Computational Thinking

Using Computer Game Programming to Teach Computational Thinking Skills

Linda Werner, *University of California, Santa Cruz, California, U.S., linda@soe.ucsc.edu*
Jill Denner, *ETR, Scotts Valley, California, U.S., jilld@etr.org*
Shannon Campe, *ETR, Scotts Valley, California, U.S., shannonc@etr.org*

Key Summary Points

1 Computer game programming can be used to engage middle school students in the development of computational thinking skills.

2 This paper describes a framework, Game Computational Sophistication, which is used to evaluate students' games regarding their computational sophistication.

3 Best practices include suggested assessment strategies, and ways that teachers can use computer game programming to maximize computational thinking.

Key Terms

Computer game programming
Computational thinking
Metrics for operationalization of computational thinking
Computational sophistication
Programming construct
Pattern
Game mechanic
Middle school

Introduction

A good way for teachers to motivate students to work on computational thinking (CT) skills is by bringing computer game programming into the K-12 classroom. CT is described as a set of skills that includes formulating problems, logically organizing and analyzing data, representing data through abstractions, and automating solutions (Barr & Stephenson, 2011). Selby (2013) proposes a definition of CT focusing on the activities that develop acquisition and provides evidence of CT skills. These include the ability to think in abstractions, generalizations, algorithmically, and in terms of decomposition and evaluation.

Wing (2006) explains that "(c)omputational thinking will be a fundamental skill used by everyone in the world by the middle of the 21st century." The Computer Science Teachers Association (CSTA) has included elements of CT in its "K-12 Computer Science Standards," such as problem solving, algorithms, data representation, modeling and simulation, and abstraction (CSTA Standards Task Force, 2011). These standards also identify a developmental progression in these skills. For example, a middle school level understanding of abstraction involves being able to decompose a problem into sub-parts, whereas a high school level understanding of abstraction involves using procedural abstraction, object-oriented design, and functional design to decompose a large-scale computational problem.

While most agree that CT is a set of important skills to develop, there is little guidance on how to teach them. Lee et al. (2011) describe an instructional progression that includes the steps that teachers can take to engage students in CT and involves creating models and simulations, as well as designing and programming computer games. Selby (2013) suggests that the following activities can lead to the development of CT skills: problem solving, systems design, automation, modeling, simulation, and visualization. Our own and others' research suggests that the design and building of computer games, if done with appropriate guidance and appropriate game development tools, leads children to develop and show evidence of the use of CT skills (Denner & Werner, 2011; Denner, Werner, & Ortiz, 2012; Werner, Denner, Campe & Kawamoto, 2012; Werner et al., 2012; Repenning, Webb, & Ioannidou, 2010; Resnick et al., 2009).

In this chapter, we describe how making a computer game can engage middle school students in CT. We offer a framework that we developed to evaluate students' games for CT, and include examples of how to identify different aspects of CT in specific games, such as problem solving, algorithms, modeling, and abstraction. Finally, we describe best practices that instructors can use to increase the likelihood that computer game programming will involve CT.

Key Frameworks

The research in this chapter builds on prior studies in the areas of complex problem solving and novice programming. The creation of computer games can be a complex problem solving activity and one that young students are capable of doing. Designing and programming a game is what Jonassen (2000) has described as a "design problem" that is ill-structured, requiring the student to define the goal, the solution path, and how to evaluate the solution. For example, most games include the key features of complex problem solving that were identified by Quesada, Kintsch, & Gomez (2005). These include tasks that are: 1) dynamic (each action changes the environment), 2) time dependent, and 3) complex (requires a collection of decisions that determine later ones). To study these features, research must look at how students attempt to solve problems—what they do when they are faced with situations that are dynamic (each action changes the environment), time dependent (use timers to enhance the gameplay experience) and complex (decisions made early in the game determine later decisions).

Historically, the first programs students create are not considered complex systems since they are not dynamic, not time dependent, and not complex. These first programs typically do not focus on the user of the program. Instead, these programs implement small, but highly constrained, computational tasks, such as adding integers or displaying the words "Hello World." With the advent of powerful, yet simple-to-use, novice programming environments such as *Alice and Scratch*, young students can create their own dynamic systems—computer games—and in doing so, the students focus on the user or game player, of their creations.

Our effort to understand what children learn by programming games is based on decades of studies. For example, research on the development of programming knowledge has described developmental progressions. Both Linn (1985), with her "chain of accomplishments" example, and Robins, Rountree, & Rountree (2003) describe three dimensions that can be used to distinguish between effective and ineffective computer programming novices:

1. **Knowledge:** The knowledge of design, language, and debugging tools;
2. **Strategies:** The strategies for design, implementing the program using a programming language, and debugging; and
3. **Models:** The mental models of the problem domain, the desired program, and the actual program.

These three dimensions—knowledge, strategies, and models—provide a useful framework for identifying the types of thinking that a student engages in while programming. While these dimensions sometimes overlap, Robins et al. (2003) suggest thinking of them as stages in the process of acquiring programming skills, and within each stage, students progress through the phases of designing, generating, and evaluating their program.

Research on children programming games and digital stories has focused less on progressions and more on the computer programs the children create. These efforts typically focus on the use of programming constructs, which are one of the fundamental computer science building blocks that are accessible to students in novice programming environments (Denner & Werner, 2011; Brennan & Resnick, 2012). Most of these studies have summarized which programming constructs appear in students' final programs, but do not distinguish between programming constructs that have been successfully or unsuccessfully used. The analysis of computer programs created by children done by Werner, Campe, & Denner (2012) is important because it relies not only on the presence of a programming construct, but also analyzes its use. This analysis determines whether the programming construct is reachable along some program path and whether the construct, when executed, causes abnormal program execution.

We propose a new framework for analyzing how children develop CT skills during computer game programming called "Game Computational Sophistication" that has been informed by the work by Jonassen (2000), Quesada et al. (2005), Linn (1985) and Robins et al. (2003). This framework emerged from our analysis of student games, and accounts for multiple levels of complexity that go beyond programming constructs to look at whether game programmers are creating complex systems. At the simplest level of the framework, are the elementary code pieces of students' games or programming constructs. These include a programming language's instruction set, and what are typically described in studies of how computer game programming can teach students higher order thinking.

At the next level of computational sophistication, students put together multiple programming constructs to create instances of "patterns," which are higher order computer science building blocks that use combinations of programming constructs. Patterns create additional program functionality but may or may not be contiguous segments of code. Expert programmers have libraries of these patterns, sometimes called "plans," from which to build their programs (Brooks, 1977; Pea & Kurland, 1984; Jeffries et al., 1981; Ehrlich & Soloway, 1984). Software engineers call these plans "design patterns," based on the work by Alexander (1997) who writes they "provide a common vocabulary for design, they reduce system complexity by naming and defining abstractions, they constitute a base of experience for building reusable software, and they act as building blocks from which more complex designs can be built (Gamma et al., 1993)." It is suggested by Kreimeier (2002) that game developers "make a sustained, conscious effort to define and describe the recurring elements of their daily work ... so we can begin to create software tools made or adapted specifically for game design purposes." The identification of game design patterns creates a common language for both designing and analyzing games (Holopainen & BjÖrk, 2003). Repenning and his colleagues describe patterns at the level of phenomena (e.g., collision, transport, and diffusion), and they explore whether students can transfer the use of those patterns to other applications (Ioannidou, Bennett, Repenning, Koh, & Basawapatna, 2011). While these authors have advanced our understanding of how to think about and identify patterns, studies examining the incomplete, successful, and unsuccessful patterns used to create games developed by middle school youth are nonexistent.

At the highest level, the game computational sophistication includes "game mechanics," which are a combination of programming constructs and patterns. They are used to make the game fun to play and to challenge the player. Game mechanics are the actions, behaviors, and control mechanisms that are available to the player (Hunicke, LeBlanc, & Zubeck, 2004) and provide the kinds of actions that the player must take to move gameplay along. Sicart (2008) provides a definition of game mechanics that is useful for game analysis: "methods invoked by agents, designed for interaction with the game state… something that connects players' actions with the purpose of the game and its main challenges." In other words, the game designer must engage in complex problem solving to create rules, interactions between the rules, and the mechanics (the game pieces that provide the interactivity for the player) to address a challenge or set of challenges within the game. We know of no studies of games that identify game mechanics in games developed by youth. Discussions with game design experts and researchers have advanced our understanding of how to think about and identify game mechanics. Similar to the research on programming constructs and patterns, we are not familiar with any research that has examined the properties of incomplete, successful, and unsuccessful game mechanics in games developed by youth.

Key Findings

In this section, we describe how we used the Computational Sophistication framework to understand how computer game programming can teach children computational thinking skills. To assess the computational sophistication of the students' games, we first identified the programming constructs, patterns, and game mechanics that are possible given the programming environment used, and then analyzed the games' program codes for instantiations of these three types of computer game building blocks. The differences lie in the number and computational sophistication of the programming constructs and patterns used, the number of mechanics, as well as the complexity of the integration of constructs into patterns, patterns into mechanics, and the integration between the mechanics.

The study took place in technology elective classes during or after school at seven public schools in California. Three hundred and sixty-five middle school students using the *Alice* programming environment made the games. Over a two-year period, we offered our entire *Alice* curriculum 16 different times, each over a semester. Classes were randomly assigned for students to work on their games in a pair or by themselves. Students spent approximately ten hours learning to use *Alice* by following worksheets with step-by-step instructions to introduce programming constructs, and another ten hours programming their games. Students chose the content of their games with the limitations being that the content is appropriate for school, as defined by their teacher; that the game is interactive, has a player outcome, and includes player instructions. A total of 231 games were created.

The games were analyzed for the following *Alice* programming constructs, presented in order from least to most sophisticated: *do in order* statement, *do together* statement, simple event handlers, built-in functions, *set* statement, more sophisticated event handlers, student-created methods, student-created and non-list variables, *if/else* statement, *loop* statement, *while* statement, student-created parameters, student-created functions, student-created list variables, *for all in order* statement, *for all together* statement, nested *if/else* statement.

For patterns, we identified the following 15 patterns in the student-created games, again listed from least to most sophisticated (see Table 1). The last column shows the percentage of the 231 games that included each pattern.

Table 1. Patterns

Pattern	Pattern Description	%
Parameters	Setting parameters such as font size, as seen by (but not duration) available for all built-in methods	35.5%
Sound	Use of audio sounds not built into Alice methods	13.9%
Movement	Controlling object or camera movement with key or mouse	19.5%
Manipulating subparts	Programming subparts of an object to change during the game (e.g., arm of one character hits another and just their head falls off)	25.5%
Instructions	Instructions are programmed via 3D text, methods	71.9%
Phantom objects	Using not-in-view objects to move and position other objects	4.3%
Embedded methods	Student-created method that is embedded within another method	27.3%
Dialog box	Player is asked for input, input is read in, and program uses the input	15.6%
Vehicles	Vehicle property is used so that when the vehicle object moves, an attached object moves in unison with it	21.2%
Collision	There is a program action depending on the distance one object is from another	21.2%
Camera control	Changing the view according to movement or player input within one scene	39.4%
Scene change	Programming movement to and from different scenes	12.6%
Counters	Integer variable created and initialized, variable's value incremented or decremented, and threshold value of variable triggers additional action	7.8%
Timers	Integer variable created and initialized, variable's value changed as time passes, and threshold value of variable triggers additional action	9.5%
List processing	List variables are created and used with For all in order or For all together	2%

We identified the following 11 game mechanics in the student-created games (see Table 2) based on discussions with game design experts and researchers (A. Sullivan, G. Smith, T. Fristoe & L. McBron, 2011) and by analyzing the students' games. We have found that there was a range of computational sophistication, based on programming constructs and patterns used, to build each of these game mechanics. The last column shows the percentage of games that included each game mechanic.

Table 2. Game mechanics

Game Mechanic	Game Mechanic Description	%
Collecting	Player attempts to accumulate objects to advance in game.	19.9%
Timed Challenge	Player is given a time limit to complete game task.	11.3%
Exploration	Player moves an object or the camera to find objects beyond player's initial range of view. Movement is not restricted to occur along a designated path.	13.0%
Shooting	Player shoots at object; actual projectile must be present.	2.6%
Racing	Player moves object across a finish line within time limit or moves an object in competition with other objects.	3.9%
Guessing	Player answers questions via clicking, typing, or moving an object.	22.9%
Hidden Objects	Player searches for an object that is hidden either beyond view or "hidden in plain sight."	6.1%
Navigation	Player moves object and/or camera from one location to another known location often on a designated path.	16.5%
Levels	Player moves between at least 2 stages by gathering points or fulfilling a challenge.	2.2%
Avoidance	Player moves object to avoid either stationary or moving obstacle based on player proximity to obstacle. Feedback to proximity is required.	3.5%
Hitting Moving Objects	Player attempts to click on moving object or moves something (character, object, camera) closer to a moving target to prompt another action.	5.6%

To illustrate what our Computational Sophistication Framework looks like when applied to games, specifically to illustrate a range of sophistication in what these patterns and mechanics look like, we have included two case studies (see case study section).

Case Study: *M808 Super Battle Tank*

One of the more computationally sophisticated games created by the middle school students in our study was made by a pair of boys, titled *M808 Super Battle Tank*. The students use eight unique patterns to implement three different game mechanics (Collecting, Timed Challenge, and Exploration). The student programmers use two additional patterns to enhance the visual aspects of the game. The game instructs the player to drive a tank around a city (the Exploration game mechanic) to find and destroy seven cars by clicking on them to start fires (the Collection game mechanic) within a particular time limit (the Timed Challenge game mechanic). A “win” message appears if the player destroys seven cars within the allotted time; a “lose” message appears if the time runs out and seven cars are not destroyed.

In Table 3 are listed each of the patterns used to implement each of the game mechanics found in *M808 Super Battle Tank*. To demonstrate the detail collected during our analysis, Table 3 also includes the more sophisticated programming constructs that students used to implement patterns for their Collecting game mechanic. The programming constructs have been italicized in the Collecting Game Mechanic column.

The “Instructions” pattern is part of this game’s three game mechanics since the instructions are needed to inform the game player what items to collect (part of the Collecting game mechanic), inform the game player that only three minutes are given to complete the collecting (part of the Time Challenge game mechanic), and inform the game player to move around the scene to find the cars (part of the Exploration game mechanic).

Table 3. The integration of patterns and mechanics in *M808 Super Battle Tank*

Pattern	Collecting Game Mechanic	Timed Challenge Game Mechanic	Exploration Game Mechanic
Instructions	Destroy 7 cars	Destroy 7 cars within 3 minutes	Move around city to see cars
Vehicles			Camera using tank as vehicle
Camera Control			Player seeing back of tank while moving around game scene
Embedded methods	Blow up cars, counting, etc.	Check count of how many cars are blown up, win and lose messages, etc.	
Phantom objects	Placement of instructions	Placement of instructions	Placement of instructions
Timer		Destroy 7 cars within time limit	Move around to destroy 7 cars
Counter	Count the number of cars collected (i.e., clicked on) as you move through scene. *Uses variables, student-created methods, simple and more sophisticated event handlers, set statement, and built-in functions.*	Destroy 7 cars within time limit.	
Parameters	Car blowing up style is abrupt; for look and feel		
Manipulating subparts	Tank's turret is turned; for look and feel		
Key/mouse control	Tank's turret is moved. *Uses simple event handlers.*		

Case Study: *Fishy Attack*

Fishy Attack, made by a girl working alone, is a game showing a mid-range level of computational sophistication. It has two game mechanics, "Collecting" and "Timed Challenge," which the student implemented using four distinct patterns (see Table 4). The student programmed the Timed Challenge mechanic using only the Instructions pattern. The student programmed a monkey to give instructions using the *say* built-in method call and modified the duration parameter's default value of the say to keep the instructions on the screen for five seconds giving the player more time to read each of the instructions. The student also programmed a *print* programming construct that persistently displays "click on all the fishy…" below the game scene. It is important to note that the student used simple event handler programming constructs to make the fish invisible when collected. The use of simple event programming to accomplish this collecting does not constitute the use of a pattern.

Table 4. The integration of patterns and mechanics in *Fishy Attack*

Pattern	Collecting Mechanic	Timed Challenge Mechanic
Instructions	Click on all fish. Uses simple event handlers.	Click on all fish within 40 seconds.
Embedded methods	All remaining fish sink underwater.	
Timer	Click on all fish within 40 seconds.	
List processing	All remaining fish sink underwater in unison.	

The opening screen shot for *Fishy Attack* is shown in Figure 1. After the monkey on the island says, "Can you please help me get off this island," the player is instructed to "click on all the fishy" (the Collecting game mechanic) before they drown (the Timed Challenge game mechanic). As the player clicks on fish, they disappear and are saved. Unfortunately, there is no code for one of the fish to disappear when the player clicks on it; therefore, there is no way to *win* by saving all the fish from drowning. It is unclear if this was the intent of the student. When the time runs out and the player has not succeeded in saving all of the fish from drowning, all the unsaved fish *sink* underwater.

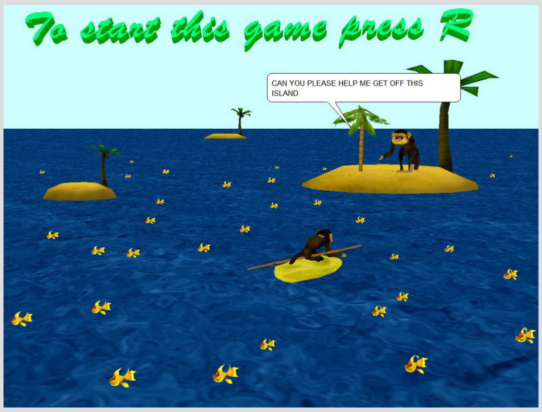
To start this game press R
CAN YOU PLEASE HELP ME GET OFF THIS ISLAND

Figure 1. Opening screen shot for Fishy Attack

Assessment Considerations

Game-based assessment techniques such as we have described with our game computational sophistication framework provide only one strategy for measuring computational thinking skills. Their contribution is that they allow a quantifiable measure of definable aspects of CT, and we can say with reasonable confidence that the students engaged in those aspects. The games themselves cannot tell us how deeply the students engaged in those aspects of CT, however, or why the students included or did not include certain features—whether it was due to the complexity of the programming construct or pattern, or to a lack of interest in having that particular feature in their game. A more comprehensive picture of CT skills requires additional assessments, such as a test of students' knowledge transfer, or the collection of more in-depth, qualitative data from both students and teachers.

For example, Werner et al. (2012) measured transference of CT skills with the Fairy Assessment, which is an *Alice* game that students play solving increasingly more sophisticated CT problems by adding, debugging, and modifying the *Alice* programming code. More than 300 middle school students' solutions were scored resulting in a range of CT skills. Administration of the assessment was not costly; however, scoring of the solutions was time-consuming. Burke & Kafai (2012) analyzed *Scratch* programs created by ten inner city middle school youth enrolled in a digital storytelling class. Regarding CT skills, they found widespread use of concepts such as loops and event handling but only limited use of the more sophisticated programming concepts such as conditionals, Boolean logic, and variables. Limitations include concerns about what students were able to do on their own without help from others. Additionally, their study involves only a small number of students.

In another example, Repenning et al. (2010) have begun the analysis of games students have created using *AgentSheets* looking for the presence of CT skills. Middle and high school teachers involved in their projects report high student engagement. Limitations include whether demonstrated CT skills are transferable. The researchers have identified next steps such as to show that the students' game building skills are transferable to other areas of STEM education. The researchers have built an inventory of higher-level CT patterns used in game development. Their next step is to show use of these patterns in computational biology and chemistry simulations and robotics applications.

Brennan & Resnick (2012) have developed the most comprehensive assessment package for *Scratch* projects. This consists of three parts: 1) Automated project portfolio analysis, 2) Interviews about artifacts created, and 3) Design scenario-based testing. These researchers have identified limitations of this assessment package, repeating concerns of what students are able to do on their own when looking at the results of the automated project portfolio analysis. They reported the interview portion of the assessment is time-consuming, taking one to two hours per interview. Additionally, the researchers believe this portion of the assessment package would benefit from multiple interviews per student occurring progressively during the project development period. The design scenario part of the assessment package, similar to the Fairy Assessment described above, is time-consuming in delivery.

Future Needs

Computer game programming can teach CT skills, and we have begun to identify the kinds of computational thinking that middle school students engage in while making their personal choice of games with the *Alice* programming environment. There are limitations to our work, such as:

1. The Computational Sophistication Framework was developed by analyzing games created in Alice and needs to be tested on games created with other tools to see if the distinction between constructs-patterns-mechanics makes sense and to see if other patterns or mechanics emerge.
2. The findings need to be compared against other measures of CT collected from the same students to ensure their reliability.
3. The findings do not contribute to efforts to understand CT learning progressions, and further work is needed to determine whether certain patterns (or mechanics) are more sophisticated than other patterns (or mechanics) and whether there is a range of sophistication in how patterns or mechanics are used.
4. For this approach to be used by teachers, the assessment and analysis needs to be automated.

Case Study: *Scratch* as a Path to Programming (written by Lucas Crispen and Elizabeth LaPensée)

Scratch (scratch.mit.edu/) is a graphical programming language and development environment that is an accessible, effective, and engaging way to teach coding. It has been particularly accessible for middle school and high school students at the Self-Enhancement Academy Inc. (SEI), a non-profit organization supporting disadvantaged youth through a full-time middle school and after-school program. This case study describes the application of *Scratch* in a programming class at SEI taught alongside a partnership with Pixel Arts Game Education (www.gameeducationpdx.com/), a non-profit dedicated to reducing the barriers of access to game development technology and education. Experiences with Scratch are based on three middle school classes and one high school class taught across Fall 2013, Winter 2014, and Spring 2014 with individual class sizes ranging between five and fifteen youth.

Initially, Lucas Crispen—a game programmer with professional industry experience and academic experience in teaching and developing curriculum for weekend and summer classes and camps in digital media and game programming—was brought in to teach a general coding class. SEI selected Code Academy (www.codeacademy.org) due to its robust curriculum, and while it is excellent overall for teaching Javascript and web design, it failed to meet the needs of SEI's youth. Foremost, youth faced a learning curve since they had little to no prior programming experience, brought on by limited computer access outside of SEI classrooms. Many youth were intimidated by screens of code and self-defeating when encountering issues.

Based on these concerns, as well as a desire to better engage youth in an after-school programming class with no mandatory attendance or grade system, Crispen developed a curriculum around the visual programming environments *Scratch* and *SNAP* (a visual drag-and-drop programming language, snap.berkeley.edu). He noticed an immediate improvement in the engagement level of youth as well as the speed with which they were able to pick up basic programming concepts.

The curriculum involves nine weeks of two one-hour sessions each week, beginning with open-ended discussions about programming and simple exercises in *SNAP* and *Scratch*. In Weeks two and three, youth learn how to manipulate sprites, learn about the 2D coordinate system by drawing shapes and patterns with the pen tool, and engage in simple conditionals and loops while making a simple line-based *Snake*-like game with user input. Week 4 invites experimentation and excites youth by encouraging "hacking." The students play games from the *Scratch* community, identify how these games function based on previous lessons, and then "hack" the code of these games to adjust the difficulty level and/or change graphics or sound, which is well-supported by Scratch's "Remix" functionality.

The remainder of the curriculum reinforces core concepts including compound logic, multi-case conditionals, and conditional loops as youth make their own maze games and elevate to making their own versions of *Flappy Bird*, through cycles of development, playtesting, and iteration with other youth in the class. Youth were especially engaged by contributing to the *Flappy Bird* "clone" community and reinforced skills established earlier.

When using *Scratch* in programming curriculum, there is room for improvement in terms of performance. *Scratch* has performance issues on older computers, which is a concern for institutions and organizations with restricted technology funding. The browser version of *Scratch* also requires reliable Internet connections and speed. This can result in frustration for youth and for instructors working within limited class time.

Overall, *Scratch* is successful in achieving STEM outreach by establishing concepts and enthusiasm reinforced by integrating popular games throughout curriculum. *Scratch*'s visual nature avoids many of the language difficulties associated with learning traditional programming and allows students to focus on developing computational thinking skills and understanding core concepts. From a game development perspective, it provides an easy introduction to handling keyboard and mouse inputs, as well as a simple sprite-based system for drawing objects on the screen.

Since *Scratch* does not currently convert visual programming to existing programming language, it is best implemented as a path to understanding foundations that can be followed-up by a tool like *Stencyl* (www.stencyl.com/), which is currently used in the game development classes by Pixel Arts Game Education. Youth in the programming classes are able to directly correlate their experience designing a game with the classic *Snake* mechanic, a maze game, and a *Flappy Bird* clone to more advanced steps for designing their own self-determined games.

Best Practices

Based on our findings (Campe, Denner, & Werner, 2013), the following principles should guide teachers on how to use computer game programming to develop and engage students in computational thinking skills:

1. **Curriculum:** Schedule technology modules into your class. The entire *Alice* curriculum fits well into one semester's schedule of four hours of class meetings per week.
2. **Technology:** Choose one of the novice programming environments (Kelleher & Pausch, 2005). *Alice* and *Scratch* are the most popular and the CSTA publishes lists of resources for both of these programming environments for teachers to use in their K-12 classrooms.
3. **Teacher Prep:** Understand the range of computational sophistication involved in making different types of games using tables such as those we have given in this chapter for patterns. Understand the types of games that same-age students are interested in making to assist students in determining personal interest (Denner, Ortiz, Campe, & Werner, 2014).
4. **Pedagogy:** Guide the students to make the more sophisticated types of games.
   For example:
   a. Provide examples of more sophisticated games made by same-age students.
   b. Provide scaffolding to students for learning the novice programming environment and learning key constructs and patterns for game design and creation (Campe et al., 2013; Campe, Werner & Denner, 2012; Webb & Rossen, 2013).
   c. Guide students to design and create a practice game first. This activity motivates students to learn more sophisticated programming constructs, patterns, and game mechanics.
   d. Include student, teacher, and peer review activities of students' games to provide feedback highlighting game functionality and usability issues (such as that seen in the second case study with a "no win" situation). These can be done as group, pair, or individual activities and can be done at various points during the game development process.

Resources

Websites and Reports

National Research Council. *Report of a Workshop on the Scope and Nature of Computational Thinking.* Washington, DC: The National Academies Press, 2010.

National Research Council. *Report of a Workshop on the Pedagogical Aspects of Computational Thinking.* Washington, DC: The National Academies Press, 2011.

CSTA/ISTE CT resources (https://csta.acm.org/Curriculum/sub/CompThinking.html)

Alice website (http://www.alice.org/)

Scratch website (http://scratch.mit.edu/)

SNAP website (http://snap.berkeley.edu/)

References

Alexander, C. (1979). *The timeless way of building* (Vol. 1). Oxford University Press.

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.

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.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies, 9*(6), 737-751.

Burke, Q., & Kafai, Y. B. (2012, February). The writers' workshop for youth programmers: digital storytelling with scratch in middle school classrooms. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 433-438). ACM.

Campe, S., Denner, J., & Werner, L. (2013). Intentional computing: Getting the results you want from game programming classes. *In Journal for Computing Teachers.* Retrieved on September 8, 2013 from http://www.iste.org/store/product?ID=2850

Campe, S., Werner, L., & Denner, J. (2012). Game programming with *Alice:* A series of graduated challenges. In P. Phillips (Ed.), *Special Issue Computer Science K-8: Building a Strong Foundation.* Computer Science Teachers Association.

CSTA Standards Task Force. (2011). K-12 computer science standards. Retrieved on January 5, 2014 from http://csta.acm.org/Curriculum/sub/CurrFiles/CSTA_D-12_CSS.pdf.

Denner, J., Ortiz, E., Campe, S., & Werner, L. (2014). Beyond stereotypes of gender and gaming: Video games made by middle school students. In H. Agius & M. Angelides (Eds.), *Handbook of Digital Games.* Institute of Electrical and Electronic Engineers.

Denner, J., & Werner, L. (2011, April). Measuring computational thinking in middle school using game programming. *Annual Meeting of the American Educational Research Association.* New Orleans, LA.

Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education, 58*(1), 240-249.

Ehrlich, K. & Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. *Human Factors in Computer Systems.* Norwood, NJ: Ablex Publishing Co.

Game mechanics (n.d.). Retrieved on June 25, 2013 from the Wikipedia Web site: http://en.wikipedia.org/wiki/Game_mechanics.

Game Mechanics (n.d.). Retrieved on June 25, 2013 from the Gamification Wiki: http://gamification.org/wiki/game_Mechanics.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). *Design patterns: Abstraction and reuse of object-oriented design.* Springer Berlin Heidelberg.

Holopainen, J., & Björk, S. (2003). Game design patterns. *Lecture Notes for GDC*.

Hunicke, R., LeBlanc, M., & Zubek, R. (2004, July). MDA: A formal approach to game design and game research. *In Proceedings of the AAAI Workshop on Challenges in Game AI.*

Ioannidou, A., Bennett, V., Repenning, A., Koh, K.H., & Basawapatna, A. (2011). Computational thinking patterns. *American Educational Research Association* conference, New Orleans.

Jeffries, Robin, Turner, A., Polson, P., & Atwood, M. (1981). The processes involved in designing software. In J.R. Anderson (Ed.),. In *Cognitive skills and their acquisition* (pp. 255-283). Hillsdale, NJ: Erlbaum.

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational technology research and development, 48*(4), 63-85.

Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR), 37*(2), 83-137.

Kreimeier, B. (2002). The case for game design patterns. Retrieved on March 16, 2011 from http://www.gamasutra.com/view/feature/132649/the_case_for_game_design_patterns.php?print=1.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., & Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32-37.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14*(5), 14-29.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*(2), 137-168.

Quesada, J., Kintsch, W., & Gomez, E. (2005). Complex problem-solving: a field in search of a definition?. *Theoretical Issues in Ergonomics Science, 6*(1), 5-33.

Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 265-269). ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, J., Silverman, B., & Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review. *Computer Science Education, 13*(2), 137-172.

Selby, C. (2013). Computational thinking: the developing definition. (Submitted). In, *The 18th Annual Conference on Innovation and Technology in Computer Science Education*, Canterbury, GB, 01-03 Jul 2013

Sicart, M. (2008). Defining game mechanics. *Game Studies, 8*(2).

Soloway, E. (1986, September). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM 9*, 850-858.

Webb, H., & Rosson, M. B. (2013, March). Using scaffolded examples to teach computational thinking concepts. In *Proceedings of the 44th ACM technical symposium on Computer science education* (pp. 95-100). ACM.

Werner, L., Campe, S., & Denner, J. (2012). Children learning computer science concepts via Alice game-programming. In *Proceedings of the 43rd. ACM conference on Computer Science Education (SIGCSE 2012).* Feb. 29-Mar. 3, Raleigh, N. Carolina, USA.

Werner, L., Denner, J., Campe, S. & Kawamoto, D.C. (2012). The Fairy Performance Assessment: Measuring computational thinking in middle school. In *Proceedings of the 43rd. ACM conference on Computer Science Education (SIGCSE 2012).* Feb. 29- Mar. 3, Raleigh, N. Carolina, USA.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.

Acknowledgments

Our thanks go to the teachers and administrators at our seven schools, specifically Anne Guerrero, Shelly Laschkewitsch, Don Jacobs, Sue Seibolt, Karen Snedeker, Susan Rivas, and Katie Ziparo. Thanks also to teaching assistants, Will Park, Chizu Kawamoto, and Joanne Sanchez; and to Pat Rex and Eloy Ortiz, for instructional materials design and technology support. Thanks to Dominic Arcamone, Stephen Butkus, Melanie Dickinson, Anthony Lim, and Kimberly Shannon for their game analysis work. Thanks to all of the students who participated. This research is funded by a grant from NSF 0909733 "The Development of Computational Thinking among Middle School Students Creating Computer Games." Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.