Alexander Meets Michotte: A Simulation Tool Based on Pattern Programming and Phenomenology

Article *in* Educational Technology & Society · January 2016

CITATIONS
5

READS
118

1 author:

Ashok Basawapatna
State University of New York College at Old Westbury
26 PUBLICATIONS **454** CITATIONS

SEE PROFILE

Alexander Meets Michotte: A Simulation Tool Based on Pattern Programming and Phenomenology

Ashok Basawapatna
SUNY College At Old Westbury, Department of Math, Old Westbury, NY // CIS, University of Colorado Boulder, Department of Computer Science, Boulder, CO // ashok.basawapatna@colorado.edu

(Submitted November 11, 2014; Revised March 17, 2015; Accepted May 21, 2015)

ABSTRACT

Simulation and modeling activities, a key point of computational thinking, are currently not being integrated into the science classroom. This paper describes a new visual programming tool entitled the Simulation Creation Toolkit. The Simulation Creation Toolkit is a high level pattern-based phenomenological approach to bringing rapid simulation creation into the classroom environment. Students create agent-based simulations via analogy between the real world phenomena they are trying to represent and "interacticons," which are visual animations of generic agents enacting the phenomena. This approach obviates the need for students to implement low level and often time-consuming agent behavior programming and yet, requires the use of abstraction, which is a key component of computational thinking. Initial data shows that students in the classroom can implement simulations faster using the Simulation Creation Toolkit as compared to the end-user programming level and begins to show that students can use the Simulation Creation Toolkit to create simulations through analogical reasoning. In this sense, Simulation Creation toolkit provides an initial data point into the integration of Computational Thinking activities through simulation construction in the classroom environment.

Keywords

Computational thinking, Computational thinking patterns, Pattern based visual languages, Phenomenology, STEM education, Simulation creation

Introduction

Educational end-user game programming tools

Many educational visual programming tools are aimed at lowering the barrier of entry into computer science for end-users (Kelleher & Pausch, 2005). These tools often deemphasize syntax by enabling rule-based programming through drag and drop interfaces. Evidence shows that this strategy is successful at motivating students in the area of computer science (Kelleher & Pausch, 2005; Squire, 2003).

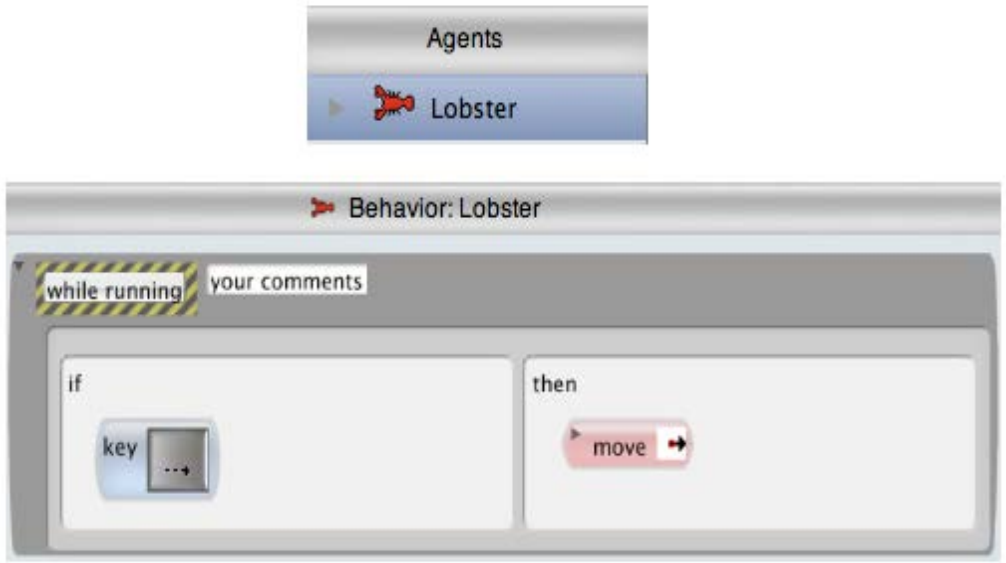

Figure 1. The top picture shows a depiction of the Lobster Agent, the bottom pictures shows one rule of the Lobster Agent's behavior which uses a "Key Pressed" condition and a "Move" action

For example, a tool we currently employ at the University of Colorado Scalable Game Design Lab is AgentCubes. AgentCubes is an agent-based rapid end-user game and simulation prototyping tool (Repenning, 2011). In AgentCubes, users can quickly create 3D games consisting of agents, which are the in-game characters. Each agent contains a depiction of how it looks and behaviors that are the set of rules that dictate its action throughout the game run. Agent behaviors rely on if/then conditionality rules wherein each rule has 2 parts: an *"if"* part containing conditions and a *"then"* part containing actions. AgentCubes currently provides users with a palette of 14 different conditions and 36 different actions that users can combine to create different agent behaviors. Figure 1 depicts a Lobster agent with one behavior rule that moves the lobster to the right when the right arrow key is hit. Previous research has shown that students can go from no prior programming experience to making their first game, Frogger, in 5 hours using AgentCubes (Repenning, Webb, & Ioannidou, 2010).

How end-user game programming relates to computational thinking and simulation design

In addition to motivating students through game design, many educational end-user programming tools also have the potential to enable "computational thinking" (Repenning, Webb, & Ioannidou, 2010). At present time, computational thinking is defined to include the following six items: problem formulation, logically organizing and analyzing data, representing data through abstractions such as models, automating solutions through algorithmic thinking, implementing effective solutions optimally, and transferring solutions to solve a large variety of problems (Barr, Harrison, & Conery 2011).

Enabling simulation and modeling activities is a means by which these tools can facilitate computational thinking in the classroom. Jeanette Wing, former Assistant Director of the National Science Foundation and a major proponent of computational thinking, states the following:

"The abstraction process—deciding what details we need to highlight and what details we can ignore—underlies computational thinking" (Wing, 2008).

In creating representational systems of real world phenomena, users choose which aspects of the real world to model based on what problem they are attempting to solve or gain insight into. Ideally, computational thinking can be achieved by enabling a user to take a problem from the real world, create a representation of this problem using an end-user programming tool, and run experiments while altering simulation parameters to get a better understanding of the real world concept being studied.

Computational thinking patterns

In an attempt to formalize the link between game design and simulation creation, we have come up with a construct entitled "Computational Thinking Patterns." Computational Thinking Patterns are agent behaviors users initially learn in game design but transfer to agent behaviors used in simulations (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). In this respect, Computational Thinking Patterns can be thought of as the "units of transfer" between game and simulation design. For example, in the game Pacman, users might learn the tracking pattern implementation to enable Ghost agents to chase after the Pacman agent. Similarly, in a predator/prey simulation a user might use the tracking pattern to have a hungry Fox agent chase after a Rabbit agent.

Computational Thinking Patterns elucidate the low-level if/then conditionality behaviors necessary in implementing various agent interactions. Table 1 lists some Computational Thinking Patterns relevant to this paper. A more exhaustive list with specific examples can be found in (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011).

Previous research has shown that when users learn a given Computational Thinking Pattern in a game context, they can effectively identify this same pattern in other contexts (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). Furthermore, there is ongoing current research dedicated to assessing games through the identification of the Computational Thinking Patterns contained within (Koh, Basawapatna, Bennett, & Repenning, 2010).

Table 1. Example computational thinking patterns

Change	One agent changes into another agent.
Absorb	One agent makes another agent disappear.
Transport	One agent transports another agent.
Push	One agent pushes another agent.
Random movement	An agent moves randomly.
Tracking	One agent chases another agent.
Keyboard movement	Keyboard button presses control an agent's movement.
Directional movement	An agent moves in a single direction at a certain speed.
Generate	One agent creates another agent.
Data	Counts the number of agents in existence at a particular instance of a simulation run.

Present barriers to classroom simulation and modeling activities/motivation for a new tool

At the present time there are many government and non-government organizations promoting computational thinking integration into the classroom environment. Organizations such as the President's Council On Technology, the National Science Foundation, the Computer Science Teachers Association, the Next Generation Science Standards as well as third party groups such as the Shodor Foundation all endorse integrating aspects of computational thinking and specifically, simulation and modeling activities, into middle and high school classrooms (Basawapatna, 2012). Given the availability of accessible end-user programming tools, one might suspect that classrooms are already integrating simulation and modeling activities. However, this is not the case. A 2009 report by the U.S. Department of Education stated, in a combined category of computers/math/science, that 75% of classes rarely or never partake in in-class simulation or modeling activities (Gray, Thomas, Lewis, & Tice, 2010). This sentiment is echoed by the National Science Foundation, which states that it is increasingly difficult to integrate computer related activities in "already overburdened K-12 curriculums" (NSF Press Release, 2009).

For example, through the Scalable Game Design project at CU Boulder, we have instructed a number of teachers leading to more than 20,000 submitted student projects. These projects include games such as Frogger and simulations such as epidemiology and predator/prey. In two separate high school classrooms, we have observed that predator/prey, for example, takes one and half to two weeks to program (assuming a 50 minute class period). This includes the time it takes to gain sufficient programming skills to create the predator/prey simulation (Basawapatna, 2012). For a computer science class, with access to a computer lab, this is an acceptable amount of time to enable students with little or no prior programming experience to create simulations. However, for a Life Science class in which the emphasis is not learning the intricacies of programming, this time and resource commitment is often prohibitive.

A method to enable more rapid in-class simulation and modeling activities while preserving computational thinking could reduce the barrier to entry into Life Science and other non-computer lab based classes. Computational Thinking Patterns, in the context of AgentCubes programming, provide a higher-level pattern-based programming construct that could accomplish this goal. Currently, if students want to implement Computational Thinking Patterns, they have to do it at the low rule-based level. For example, if a student wants to implement a Fox agent tracking a Rabbit agent, they first program the Rabbit agent behaviors to emit a scent value, then have a Background agent diffuse this scent value through a complex diffusion equation, and finally implement a hill climbing algorithm in the Fox agent behaviors such that the Fox agent moves towards the highest Rabbit agent scent value each time it moves. Though undeniably educational, in a Life Science class, for example, implementing the math necessary to have a Fox agent chase a Rabbit agent is not relevant to the predator/prey unit being studied in class. In fact, students have already made the necessary abstraction at the point of pattern implementation—namely the Fox agent *tracks* the Rabbit agent. If instead there existed a mechanism such that students could specify the agents present in the tracking interaction, and the underlying if/then conditionality rules were automatically added, it could save time and yet preserve the computational thinking abstraction aspects of the simulation construction activity.

The remainder of this paper will describe the Simulation Creation Toolkit that is a first attempt at such a system. The Simulation Creation Toolkit is built on top of the AgentCubes environment, and affords users the ability to create simulations at the higher Computational Thinking Pattern level directly through analogy with the real-world phenomena they are modeling.

The simulation creation toolkit

Design principles

The aim of the Simulation Creation Toolkit is to enable users to more easily create simulations by defining behaviors at the Computational Thinking Pattern level as opposed to the lower if/then conditionality statements level. The development of this toolkit follows the four general design principles outlined in Table 2 allowing for easy student use.

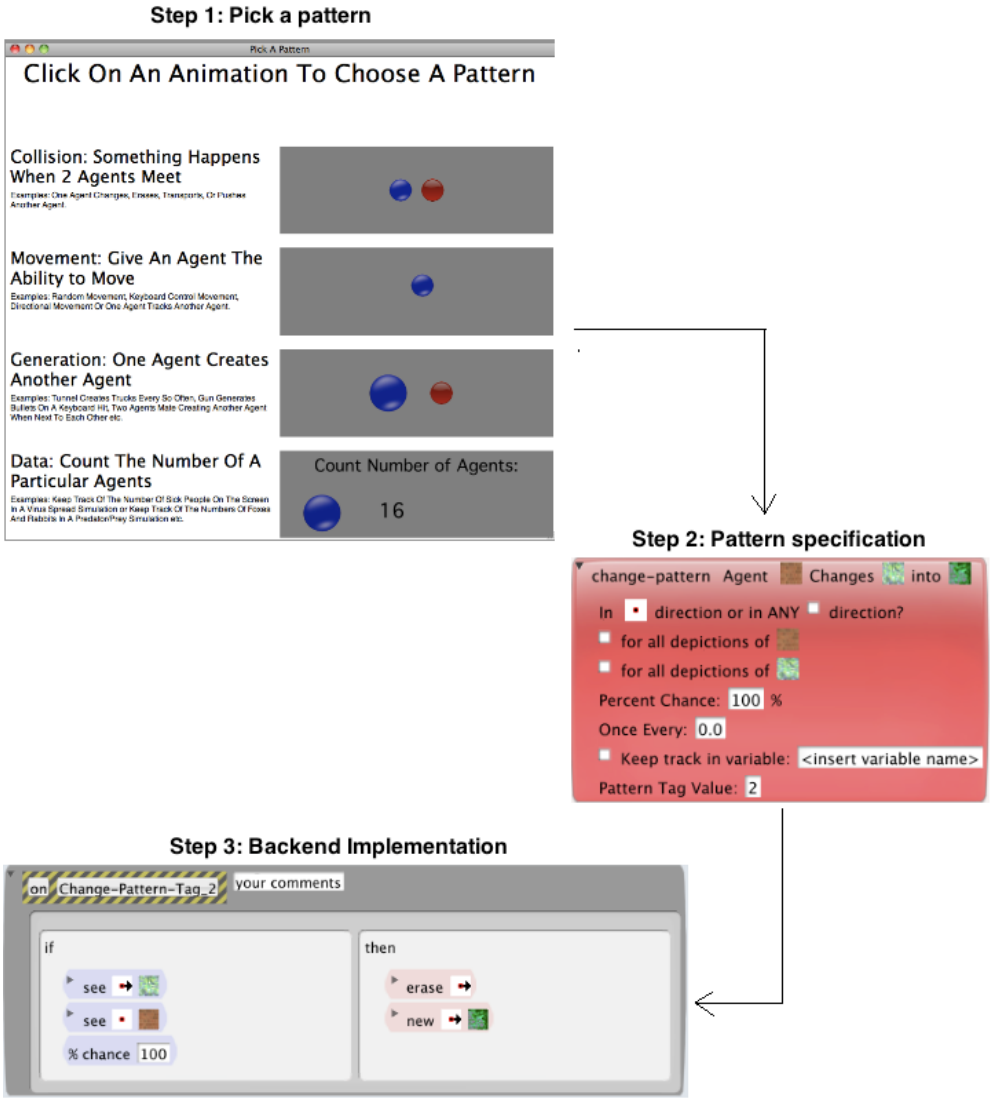

Figure 2. Implementing a pattern using the Simulation Creation Toolkit. User picks and specifies a pattern and associated if/then behaviors are added automatically

Table 2. Four design principles of the simulation creation toolkit

Number	Design principles
1	User should be able to intuitively select from a variety of patterns
2	Patterns should be customizable for a large variety of purposes
3	Every pattern should be expressed independent of the other patterns implemented
4	Implemented patterns should be easy to modify and delete

The first design principle relates to how the patterns included in the system are presented to the user. The second design principle deals with how the user specifies a given pattern, further defining under what conditions the pattern is executed. The third design principle is the assurance that every pattern the user implements is expressed during the program run, regardless of past or future implemented patterns. The final design principle ensures that users can always modify their program with ease.

Figure 2 depicts the three steps that occur as a user implements a pattern in the Simulation Creation Toolkit. Subsequent sections describe each of these steps.

Patterns and "interacticons"

The first design principle in Table 2 relates to the number of Computational Thinking Patterns available to the user and how these patterns are represented to the user. The Simulation Creation Toolkit allows users to add agent interactions by selecting from 10 different Computational Thinking Patterns, outlined in Table 1. Our research experience with more than 40 schools and 20,000 student-made projects shows that these Computational Thinking Patterns are high level patterns that are commonly implemented when creating games and simulations (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). The initial patterns included allow students to make a variety of simulations in Life Science classes including predator/prey and epidemiology. This list is by no means exhaustive and as more useful patterns are identified this palette could grow.

The first four patterns in Table 1 are "collision" patterns. These patterns occur when two agents meet in space, and include the change, absorb, transport, and push pattern. For example, the absorb pattern occurs when a Fox agent comes into contact with a Rabbit agent representing the Fox agent eating the Rabbit agent. The transport pattern and push pattern in Table 1 are collision patterns, which also happen to be "modulating" patterns. Modulating patterns change the implementation of previously modified patterns (Bjork & Holopainen, 2006). The transport and push pattern modify any movement behaviors. For example, if agent "A" were to transport agent "B" in a simulation, this transport would modify any movement rules associated with agent "A." Specifically, on the condition that agent "B" is stacked on agent "A," agent "A" carries agent "B" as it executes its movement rules. The subsequent four patterns in Table 1 are "movement" patterns which enable agents to move around a level. These include random movement, tracking, keyboard movement, and directional movement. The final two patterns in Table 1, generate and data count, do not belong in any category. The generate pattern allows one agent to create another agent. This can happen on a collision, for example when two Fox Agents mate creating a third Fox Agent, or once every so often such as when a Tunnel agent generates a stream of trucks in the game Frogger. Finally, the data count pattern enables the user to count populations of agents. In a predator/prey simulation, for example, it is common to plot the populations of the predator agents and the prey agents over time.

The user selects patterns to apply from an animated palette containing generic agents acting out each interaction outlined in Table 1. These animated representations of Computational Thinking Patterns are called "interacticons." Figure 3 depicts three sequential frames of the absorb pattern interacticon.

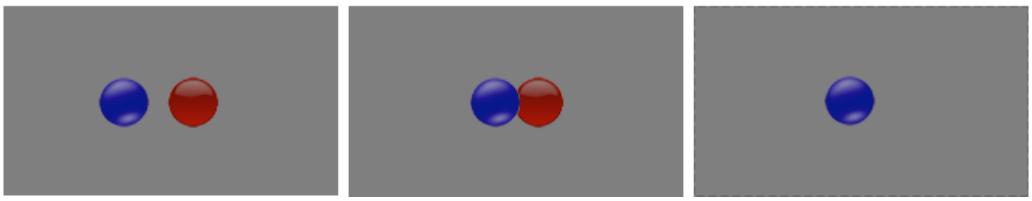

Figure 3. Three sequential frames (from left to right) of the absorb pattern interacticon

In Figure 3 the generic Blue agent (left) collides with the generic Red agent (right) with the Red agent being absorbed. Interacticons provide users with a phenomenological representation of a given Computational Thinking Pattern. The user then decides which agents from their simulation to replace these generic agents with by making an analogy from the generic interacticon representation to the real world phenomena a user wishes to represent. Figure 4 shows the same three frames of Figure 3 in the case where a user chooses a Frog agent in place of the blue disk and a Butterfly agent in place of the red disk.

Figure 4. The sequential frames of the absorb pattern depicted in Figure 2 with a Frog agent and Butterfly agent replacing the generic interacticon agents

The Simulation Creation Toolkit enables users to make simulations by combining the 10 Computational Thinking Patterns outlined in Table 1. The user chooses a pattern via analogy with a real world phenomenon.

Pattern specifications

The second design principle in Table 2 requires that patterns can be customized to fit a given interaction. After the user applies a pattern, the user specifies how the pattern will work in the simulation. Specifications covers the conditions under which the pattern executes; often these include the percent chance associated with the pattern, the frequency of a pattern, and the direction it occurs in among many other possibilities. For example, if a user implements the random movement pattern, after identifying the agent involved in the pattern, the user still needs to specify how quickly the agent will move and if this movement is blocked by other agents. Figure 5 depicts how the user specifies the random movement pattern using the Simulation Creation Toolkit.

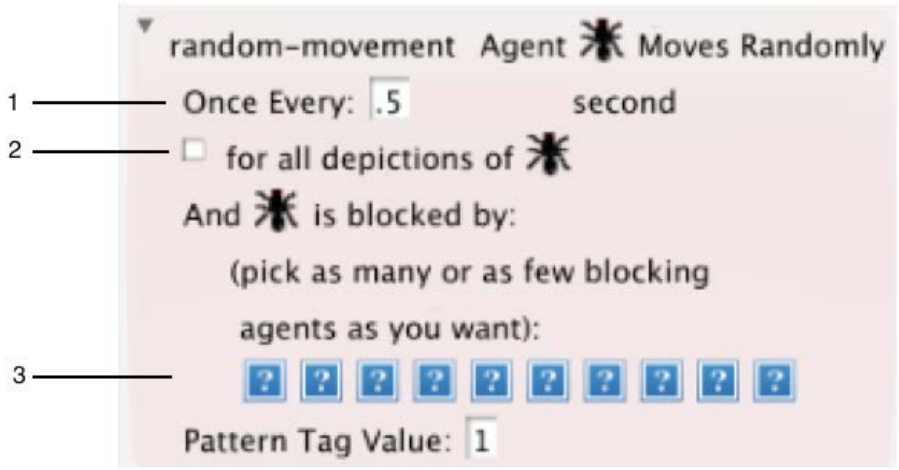

Figure 5. The random movement pattern specification

In Figure 5, the user is presented with some choices pertaining to the random movement. This includes how fast the agent moves (label 1), if it applies to all depictions of the agent (label 2), and which agents block this random movement (label 3). It should be noted that an agent can have many different depictions in AgentCubes, for example, the Ghost agent in Pacman looks different after the Pacman agent has eaten a power pellet. Different patterns have a different set of specification choices, and the choices of specifications provided to the user are tailored to how the patterns are commonly configured in our experience teaching game design and simulation activities to students.

Pattern specifications often represent system parameters. For example, the random movement specification depicted in Figure 5 enables the user to change how fast this agent movement occurs. Similarly, if a user is implementing an epidemiology simulation wherein a sick agent changes a healthy agent into a sick agent using the change pattern, the user can set a percent chance associated with this change pattern using the change specifications. In this case, the percent chance can be thought of as setting how susceptible an agent is to the infection being modeled, and this specification can be quickly changed to model different communicable illnesses.

Backend implementation

The Simulation Creation Toolkit automatically and immediately adds AgentCubes if/then conditionality code depending on the user selected pattern and associated specifications. The third and fourth design principle in Table 2 require that the system enables each implemented pattern to be executed regardless of prior implemented patterns and patterns should be easily modified and deleted. The backend of the Simulation Creation Toolkit employs various mechanisms to allow for this.

The AgentCubes architecture that the Simulation Creation Toolkit is built upon implies a priority based on rule order. Figure 6 is an example agent behavior with two rules residing in the agent's "while running" method which is the only method guaranteed to be executed for a given agent each update cycle.

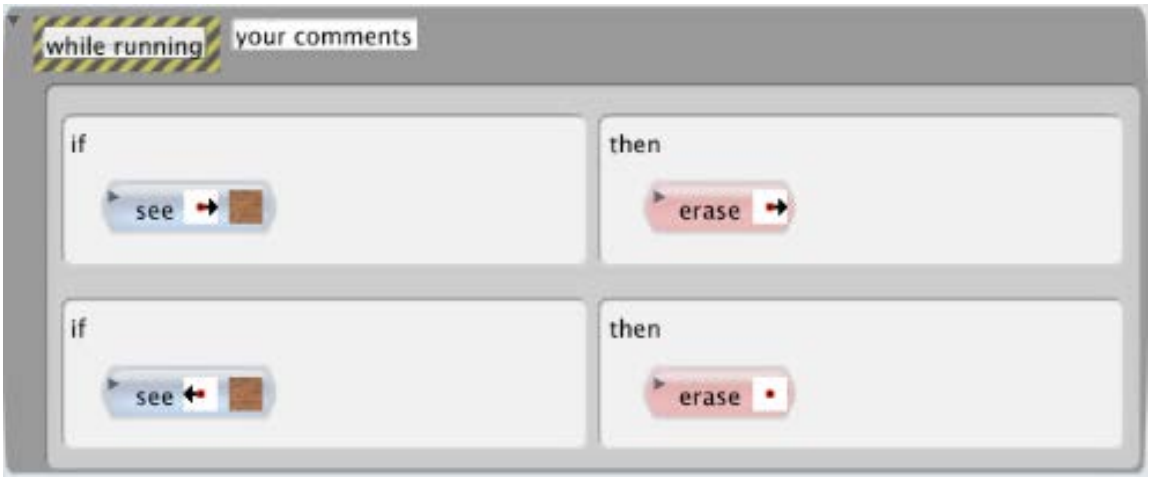

Figure 6. Example agent behavior in AgentCubes

The behavior of the agent depicted in Figure 6 is dictated by two rules. The first rule states that if this agent sees a Box agent to the right, then this agent will erase the agent to its right (deleting the Box agent). The second rule states that if this agent sees a Box agent to its left, it will erases itself. Behaviors are executed from top to bottom and only one rule from each method is executed every update cycle. Therefore, in the situation outlined in Figure 6, wherein this agent has a Box agent to its left and to its right, only the Box to its right will be erased and the agent will not erase itself in the current update cycle.

Programming agent behaviors at the lower if/then conditionality statement level affords users the ability to reorder rules based on their priority. At the pattern level, however, if a user implements two patterns, both should have a chance to be executed every cycle. The Simulation Creation Toolkit implements each pattern in its own method. Every update cycle, a single rule in the agent's while-running method calls every pattern's method enabling every pattern to be executed independent of other patterns. This ensures every pattern can be expressed during each agent update cycle.

These pattern methods also play an important role in the modification and deletion of patterns. Whenever any part of a pattern specification is modified, which can also include changing the agents present in the pattern, the method or methods that pattern resides in change immediately. The backend changes the necessary conditions and actions within the pattern method to reflect the changes to the specification. Similarly, when a user deletes a pattern, the backend deletes any method corresponding to that deleted pattern and erases any call to that method.

Previous research

Pattern languages have a rich history and have long been recognized as a way to capture high-level concepts and design practices in a variety of disciplines. Alexander et al. (1977) wrote *A Pattern Language,* outlining patterns one can implement to create towns, cities, and urban centers. Though related to a different discipline, the aims of *A Pattern Language* are similar to Computational Thinking Patterns in that each pattern can be thought of as a high-

level implementation concept that accommodates a given set of issues. Furthermore, patterns in *A Pattern Language* start with general implementations but are fully implemented through sub patterns. Patterns can have multiple implementations that each do something slightly different but are all in the general category of that pattern. The Simulation Creation Toolkit similarly defines general pattern categories (i.e., collision and movement), with various sub patterns and specifications that differentiate each of the pattern implementations.

More recently the value of pattern languages has also been recognized by software engineering. The seminal text *Design Patterns: Elements of Reusable Object Oriented Software* provides commonly used programming patterns to non-expert users (Gamma, Helm, Johnson & Vlissides, 1994). The book *Patterns In Game Design* applies this construct to the domain of game creation (Bjork & Holopainen, 2006). It provides a "Pattern Collection" which uses a similar categorization to the Simulation Creation Toolkit in terms of movement and collision patterns. It also outlines the idea behind "modulating patterns" described previously.

Interacticons play a vital role in the Simulation Creation Toolkit and are rooted in the field of Phenomenology. Albert Michotte, in his book *The Perception of Causality* devised multiple experiments to gain insight into how people perceive causal relationships even when none exist (Michotte, 1963). To this end, Michotte used devices such as timed image projections of one circle meeting another circle, and the second circle moving. With correct timing, Michotte noticed that people explained the motion of the second circle as due to contact with the initial circle. Michotte realized that the generic object motion implied a "functional relationship" between these objects. Many of the interactions Michotte describes mirror patterns present in the Simulation Creation Toolkit. For instance, Michotte describes the "Transporting Effect" wherein one object is perceived to transport another. With both Michotte's experiments and Simulation Creation Toolkit interacticons, the success or failure of generic agents exhibiting a pattern depends on the ability of people to abstract out the agents of an interaction while preserving the interaction itself.

The Simulation Creation Toolkit is an attempt to domain orient end-user game design towards simulation and modeling activities through a higher-level visual language. There are many prior instances where this practice has had success. Construction Kits, for example, supply users with high-level building blocks that they can combine to solve a specific problem (Fischer & Lemke, 1988). For example, the Pinball Construction Kit provides users with pre-programmed flippers and bumper that can be placed in a level to easily prototype their own pinball game.

Generally, these problems include helping to break the complexity barrier, utility barrier (ratio of value to energy expended), automatically taking care of time consuming and technical user tasks, all while mirroring the abstractions of the application domain (Fischer & Lemke, 1988). In this respect, the aim Simulation Creation Toolkit is very similar to that of Construction Kits. Storytelling Alice provides an example of modifying an end-user programming tool with higher-level domain oriented functionality (Kelleher, 2006). Storytelling Alice, built on top of the Alice environment, provides a suite of high-level functionality that focuses users on creating narratives. Finally, Kodu, developed by Microsoft Research, provides users with the ability to create games using high level patterns but still is not fully at the interaction level as it employs the patterns as actions in a conditionality rule (Stolee & Fristoe, 2011). Kodu includes some of the same high-level patterns like tracking and transport. One big difference between Kodu and the Simulation Creation Toolkit is the absence of interacticons.

Study design

Two studies run on the Simulation Creation Toolkit—an in class study with 45 seventh grade Life Science students creating a predator/prey simulation, and an analogical reasoning study with six university students creating three different projects—begin to give initial insight into the system capabilities. Previous research looked at how successful sixth grade computer class students with no prior programming experience could create parts of the above predator/prey simulation (Basawapatna, Repenning, & Lewis, 2013). Unlike the current study, the previous study was not integrated into the actual class curriculum; students were not in a Life Science class. Furthermore, the previous study presented no analogical reasoning evidence and students did not have the full amount of time to complete the simulation.

7th grade in-class study

The seventh grade in-class study took place over 4 days. Most of these students had brief prior experiences programming games and simulations in AgentCubes. This study investigated to what extent the Simulation Creation Toolkit could be used efficiently and effectively to create simulations of Life Science topics currently being studied. Students programmed the predator/prey simulation in their Life Science class as they were studying ecosystems and food webs. Mentioned above, in prior experiences it was observed that the predator/prey simulation takes a week and a half to two weeks to program using AgentCubes. The predator/prey simulation consists of 16 pattern implementations, outlined in Table 3. As customary with classroom simulation studies, students were provided with scaffolding materials such as a tutorial containing a worksheet of questions that they answered as they created the simulation.

Table 3. Outline of patterns present in the predator/prey simulation

Pattern	Outline
Pattern 1 and Pattern 2	The Fox Agent and Rabbit Agent start out moving randomly.
Pattern 3 and Pattern 4	The Fox and Rabbit Agent gets hungry with a given percent chance as they move.
Pattern 5 and Pattern 6	A hungry Fox or Rabbit dies with a given percent chance every second it does not find food…
Pattern 7 and Pattern 8	… and decomposes into a Grass Agent.
Pattern 9	A hungry Fox Agent tracks the Rabbit Agent.
Pattern 10	A hungry Rabbit Agent tracks the Grass Agent.
Pattern 11 and Pattern 13	If a hungry Fox encounters a Rabbit, it eats it and is no longer hungry.
Pattern 12 and Pattern 14	If a hungry Rabbit Agent encounters a grass Agent, it eats it and is no longer hungry.
Patterns 15 and 16	If a Fox or Rabbit is next to another Fox or Rabbit Agent respectively, it will mate with some percent chance.

Analogical reasoning study

The second study investigated to what extent users could analogically reason using the Simulation Creation Toolkit. Six university level students, three with prior AgentCubes experience and three without any AgentCubes experience, were given three general descriptions of requirements for programs to create. Students had as much time as they wanted to complete the program but were given no additional materials other than the descriptions and a brief five-minute introduction to the system. Table 4 provides the descriptions students received for each program – students were not provided with the "Interaction" labels in parenthesis present in Table 4, which are displayed so we can refer to each interaction later in the paper.

Table 4. Outline of patterns present in the predator/prey simulation with each interaction numbered

Interaction	Outline
Description 1: Pacman	
Interaction 1	Pacman moves with keyboard keys.
Interaction 2	All the Ghosts pursue Pacman and when they get to Pacman, Pacman disappears.
Interaction 3	Pacman eats pellets as he navigates around the level.
Interaction 4	Neither Pacman nor the Ghosts can go through the blue walls.
Description 2: Epidemiology simulation	
Interaction 1	All depictions of the person move around randomly.
Interaction 2	A healthy person has a 30% of becoming sick each second that healthy person is next to a sick person.
Interaction 3	A sick person has a 30% chance of recovery every second.
Interaction 4	A sick person also has a 10% of death (i.e., disappearing) every second.
Description 3: Predator/Prey simulation	
Interaction 1 & 2	Foxes and Rabbits move randomly.
Interaction 3	Every so often the Fox Agent gets hungry;…
Interaction 4	… at this point the Hungry Fox tracks the Rabbit Agent.
Interaction 5 & 6	If the Hungry Fox gets to the Rabbit Agent, it kills it and is no longer hungry.
Interaction 7	Eventually the dead Rabbit decomposes.

Interaction 8	Hungry Foxes can also sometimes die of Hunger.
Interaction 9	Finally, Foxes sometimes reproduce with other Foxes creating a new Fox at some percentage; …
Interaction 10	… Rabbits sometimes mate with other Rabbits creating a new Rabbit at some percentage.

Results and discussion

Seventh grade in-class study results and discussion

Figure 7 depicts a sixteen axes graph with the value at each axis representing the average percent pattern correctness among all students for that particular pattern. Patterns were scored as follows: 2 points were given if the pattern and agent were correct and 1 point was given for each specification that was correct. If every student completed the simulation entirely correct, Figure 7 would be 16-sided shape with every vertex at 100%. The lowest pattern in Figure 7 is 85% correct on Pattern 14 meaning that students, for the most part, were able to correctly implement the patterns and specifications necessary for the predator/prey simulation using the Simulation Creation Toolkit.

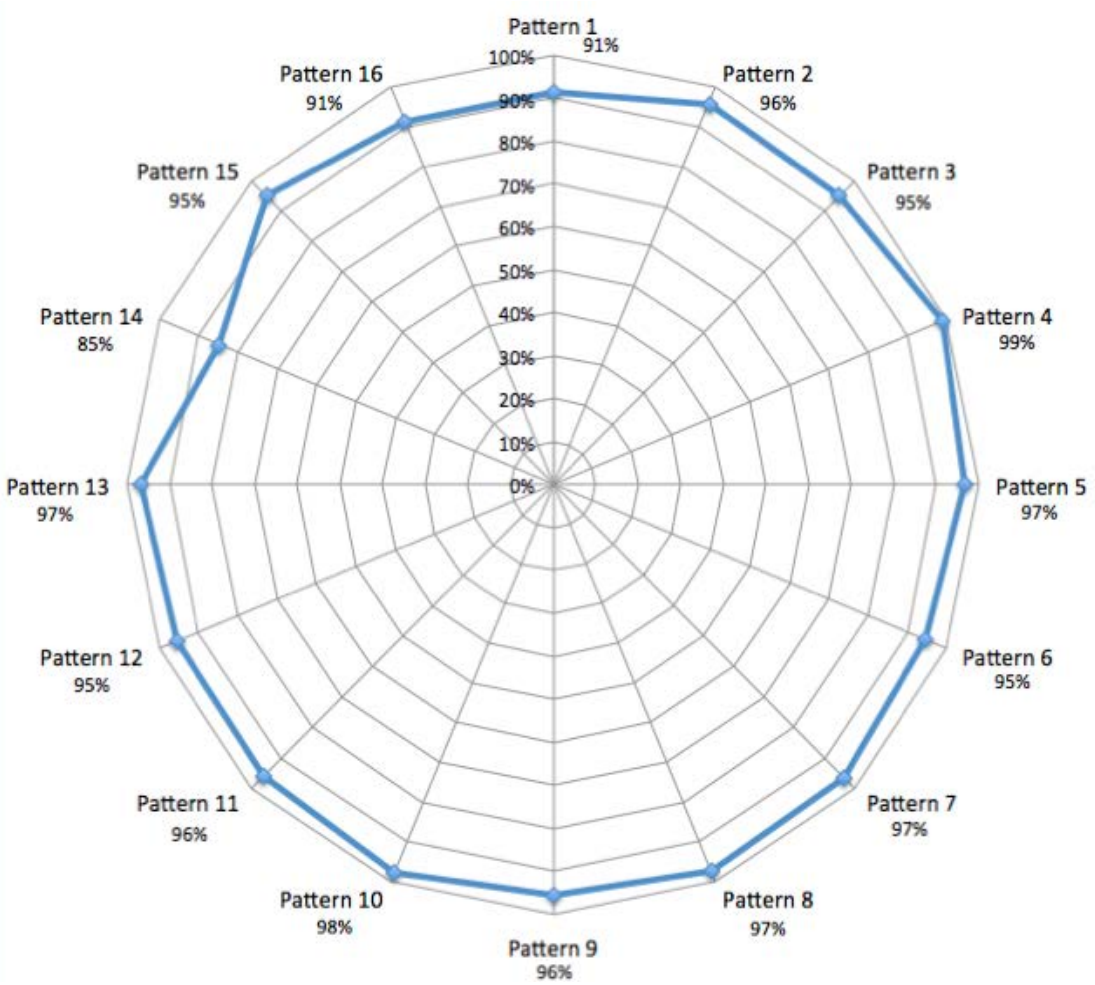

Figure 7. Average percent pattern correctness for each pattern in the predator/prey simulation among all seventh grade students

Table 5 shows the percent of students who had correct simulations, simulations with one mistake, two mistakes, and three or more mistakes. A mistake is defined as a required interaction that was not implemented or was implemented incorrectly (the pattern or a specification was incorrect or missing).

Table 5. Percent of 7th grade students with correct predator/prey simulations

	Perfect simulation	1 incorrect pattern	2 incorrect patterns	3 or more incorrect patterns
Number of students	18	9	9	9
% of students	40%	20%	20%	20%

Table 5 shows that 40% of seventh grade students were able to create a perfect predator/prey simulation in four days. Furthermore, 80% of students had two incorrect patterns or less. The above data is promising in terms of using a higher-level pattern based language to integrate simulation creation activities in the classroom. Figure 7 shows that students were able to construct the specific interactions necessary to create the simulation. In Table 5 we see that most of these students, who had no prior experience with the Simulation Creation Toolkit, were able to create working or almost working simulations in substantially less time.

Analogical reasoning study results and discussion

The analogical reasoning study looked at the extent to which users could create games and simulations given a general description of interactions (Table 4 shows which parts of the description correspond to each program interaction). Each interaction was inspected manually to see to what extent the interaction was correct. Interactions were given a "√" if they were entirely correct, a "√-" if the pattern(s) were correct but a specification was incorrect, and "X" if the pattern(s) were incorrect. Table 6 depicts the results for the Pacman program.

Table 6. Pacman simulation analogical reasoning study results

	Duration	Inter. 1	Inter. 2	Inter. 3	Inter. 4
Participant 1*	13 minutes	√	√	√	√
Participant 2*	19 minutes	√	√	√	√
Participant 4*	14 minutes	√	√	√	√
Participant 3**	20 minutes	√	√	√	√
Participant 5**	36 minutes	√	√	√	√
Participant 6**	30 minutes	√	√	√	√

Note. *Users with prior AgentCubes experience; **Users with no AgentCubes experience.

All participants created the Pacman simulation entirely correctly using the Simulation Creation Toolkit. The users with no prior AgentCubes experience seemed to take longer to create their simulations. It should be noted that in our experience students with prior AgentCubes experience can take 2 hours to program Pacman. Table 7 depicts the results for the epidemiology simulation.

Table 7. Epidemiology simulation analogical reasoning study results

	Duration	Inter. 1	Inter. 2	Inter. 3	Inter. 4
Participant 1*	15 minutes	√	√-	√-	√-
Participant 2*	9 minutes	√	√-	√	√
Participant 4*	18 minutes	√	√	√	√
Participant 3**	13 minutes	√	√	√	√
Participant 5**	40 minutes	√	√-	√-	√
Participant 6**	44 minutes	√	√	√	√

Note. *Users with prior AgentCubes experience; **Users with no AgentCubes experience.

The Epidemiology Simulation study shows that users had trouble with some pattern specifications. Still, most users identified the correct patterns. Interacticons only depict the general pattern; thus, users could make the correct analogy and pick the correct interacticon but still overlook a specification choice. Three users completed the simulation entirely correctly; two of them were novice users. The sample size is too small to draw any conclusions about novice users versus user who have had prior AgentCubes experience; however, the results in general are promising in terms of analogical reasoning with interacticons. This data also points to the possible need for depicting the specifications in a way that is more intuitive for the users. Table 8 depicts the results for the predator/prey simulation analogical reasoning study.

Table 8. Predator/prey simulation analogical reasoning study results

	Duration	Inter. 1	Inter. 2	Inter. 3	Inter. 4	Inter. 5
Participant 1*	40 min	√	√	√	√	√
Participant 2*	30 min	√	√	√	√	X
Participant 4*	32 min	√	√	X	√	X
Participant 3**	25 min	√	√	√	√	√
Participant 5**	1 hour	√	√	√	√	√
Participant 6**	1 hour	√	√	√	√	X
		Inter. 6	Inter. 7	Inter. 8	Inter. 9	Inter. 10
Participant 1*		√	√	√	√	√
Participant 2*		X	√	√	√	√
Participant 4*		X	√	√	X	√
Participant 3**		√	√	√	√	√
Participant 5**		X	X	√	X	X
Participant 6**		X	√	√	√	√

Note. *Users with prior AgentCubes experience; **Users with no AgentCubes experience.

The results in Table 8 show that users struggled with the more complicated predator/prey simulation. Certain interactions were incorrect; the biggest one was interaction 5 and 6 wherein the hungry Fox and hungry Rabbit agent change back into a regular Fox and Rabbit agent after eating. Many users stated that the interacticon for the change pattern was somewhat misleading. Namely, the interaction involves having a dead agent change a hungry agent back into a normal agent which was unintuitive. This points to the possible need for a variety of interacticon examples showing multiple different applications for each pattern rather than one generic interacticon.

General discussion

The above results indicate that novice students can use the mechanics of the Simulation Creation Toolkit effectively and in a time-efficient manner to make simulations in-class. Furthermore, the small scale analogical reasoning study gives hope to the idea of users programming simulations directly by analogy with very little scaffolding provided. In the broader educational context, this tool might be a step towards more pragmatic constructivist activities involving computational thinking in the classroom environment (Karagiorgi & Symeou, 2005).

Previous research on the Consume-Create spectrum related to classroom simulation activities provides a framework with which we can begin to analyze the efficacy the Simulation Creation Toolkit approach (Basawapatna, Repenning, Koh, & Savignano, 2014). The Consume-Create spectrum aims to enumerate simulation related activities that teachers can do in the classroom environment. Activities on the Consume end of the spectrum, such as giving students an animation or an interactive simulation, are convenient for the classroom environment as they take very little class time, however, have minimal value in terms of exposing students to computational thinking concepts. On the Create end of the spectrum, we have activities such as traditional programming and end-user programming. For example, having students iteratively create and experiment on a simulation using an end-user programming environment exposes students to computational thinking concepts, but as previously mentioned, might be impractical because of class time constraints.

The Simulation Creation Toolkit is adjacent to Construction Set Simulation/Construction Kit activities, which are slightly more on the Consumption side of the spectrum, wherein students do not program elements, but rather, place the elements in unique configurations to experiment on a given problem. For example, one can think of placing pre-programmed resistors or capacitors to make unique circuits in a circuit construction kit. Previous research shows that such strategies can have potentially huge effects in relatively small intervention times (1-2 hours) in terms of increasing deep student knowledge of an issue (Wieman, Adams, & Perkins, 2008). However, the extent to which students are thinking computationally as they do this activity is debatable (Basawapatna, Repenning, Koh & Savignano, 2014). Similarly, on the Creation side of the spectrum, the Simulation Creation Toolkit is adjacent to end-user programming tools which, as mentioned above, are effective in integrating computational thinking concepts (Repenning, Webb, & Ioannidou, 2010).

The initial Simulation Creation Toolkit results presented here indicate that the pattern programming strategy employed begins to balance student simulation construction that enables computational thinking while also preserving convenience in the classroom environment. Students, with no prior exposure to the Simulation Creation Toolkit, were able to create simulations in the classroom environment and displayed an ability to create simulations through analogy using the system. Therefore, the data indicates that the general strategy of exploiting phenomenology at the pattern level provides a viable point on the Consume-Create spectrum for further in-class exploration and may eventually be a sweet spot for teachers in terms of exposing students to computational thinking within the time and curricular constraints of their classroom.

Comparing the Simulation Creation Toolkit with another related pattern programming approach in game and simulation design illuminate the differences and contextualize these results. For example, Microsoft Kodu allows users to build games by providing some of the same high-level patterns like tracking (Stolee & Fristoe, 2011). However, the approach taken by the Simulation Creation Toolkit involves interacticons which employs the human ability to make analogies with these generic phenomenological interactions and the real world. This abstraction is a key component of computational thinking. Kodu, on the other hand, provides static icons and explanations with descriptions that guide users to select the correct pattern. Kodu also makes users program at the behavior level by making these higher-level patterns, like tracking, actions in the behavior rules. Therefore, it is up to the user to implement the pattern rules in the correct order and with the necessary associated conditions. Using the Simulation Creation Toolkit, users program purely at the pattern level—this allows for easier and quicker user implementations but at the cost of not having the freedom of pattern specifications available to users of Kodu. This presents an interesting tradeoff and future research will look at how increasing the palette of patterns and specifications in the Simulation Creation Toolkit expands user creation freedom at the cost of increasing complexity of use.

The results also begin to expose shortcomings of the Simulation Creation Toolkit that can be improved upon. For one, the analogical reasoning and in-class study seem to show that users can pick the correct patterns, but do not necessarily choose the correct pattern specifications. Developing a way to extend the power of analogical reasoning and interacticons to the specifications might be a way to alleviate this. One idea could be to have a test world where the agents not only enact the pattern but change to reflect the specifications of the pattern in real time. For example, if a student specifies that the chaser agent should move more quickly in the tracking pattern, then in the test world the student would see the chaser agent begin to move faster as it tracks the other agent. From this the user might be better able to visualize the consequence of their thinking and possibly modify the pattern specification accordingly.

Further studies should be completed to draw more concrete conclusions about how the level of abstraction corresponds to computational thinking. A study that enables students to program many in-class simulations over the course of the semester would enable the elimination of scaffolding. In this case, would students still be able to quickly model different phenomena using the Simulation Creation Toolkit? Furthermore, such a study would see if the initial analogical reasoning benefits uncovered in this study transfer to the classroom environment over the course of the semester.

Conclusion

This paper presents a new pattern-based tool, entitled the Simulation Creation Toolkit, which enables users to create simulations at a higher level by analogy through employing interactions rooted in phenomenology. Initial results are promising. Furthermore, this study is a data point towards increasing end-user programming tool effectiveness in computational thinking classroom integration. This research could one day enable universal adoption of computational thinking concepts inside the classroom environment.

Acknowledgments

Thanks to teachers Marco Cornacchione and Mark Savignano for all their help in the classroom. Dr. Alexander Repenning for his invaluable guidance and Hilarie Nickerson for helping clean up the publication. This material is covered by the University of Colorado IRB numbers 12-0548 and 12-0141. This work is supported by the National Science Foundation under Grant Numbers 0833612, 0848962, 1138526. Any opinions, findings, and conclusions or

recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern language: Towns, buildings, construction.* Oxford, UK: Oxford University Press.

Barr, D., Harrison, J., & Conery, L. (2011). Computational thinking: A Digital age skill for everyone. *Learning & Leading with Technology*, *38*(6), 20-23.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). Recognizing computational thinking patterns. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 245-250). New York, NY: ACM.

Basawapatna, A. R., Repenning, A., Koh, K. H., & Savignano, M. (2014). The Consume-create spectrum: Balancing convenience and computational thinking in stem learning. In *Proceeding of the 45th ACM technical symposium on Computer science education* (pp. 659-664). New York, NY: ACM.

Basawapatna, A. R., Repenning, A., & Lewis, C. H. (2013). The Simulation creation toolkit: An Initial exploration into making programming accessible while preserving computational thinking. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 501-506). New York, NY: ACM.

Basawapatna, A. R. (2012). *Creating science simulations through computational thinking patterns* (Doctoral dissertation). Department of Computer Science, University of Colorado, Boulder, CO. (No. AAT 3549158)

Bjork, S., & Holopainen, J. (2006). *Patterns in game design (Game development series).* Massachusetts, MA: Charles River Media Group.

NSF Press Release (2009, December). *Computer science via interactive journalism*. Retrieved from http://www.nsf.gov/news/news_summ.jsp?cntn_id=116073

Fischer, G., & Lemke, A. C. (1988). Construction kits and design environments: Steps toward human problem-domain communication. *ACM Special Interest Group on Computer–Human Interaction (SIGCHI) Bulletin*, *20*(1), 81.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. New York, NY: Pearson Education.

Gray L., Thomas, N., Lewis, L., & Tice, P. (2010). *Teachers' use of educational technology in U.S. public schools: 2009, first look*. National Center for Education Statistics, US Department of Education Report.

Karagiorgi, Y. & Symeou, L. (2005). Translating constructivism into instructional design: Potential and limitations. *Educational Technology & Society*, *8*(1), 17-27.

Kelleher, C. (2006). *Motivating programming: Using storytelling to make computer programming attractive to middle school girls* (Doctoral dissertation). School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. (No. CMU-CS-06-171)

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.

Koh, K. H., Basawapatna, A., Bennett, V., & Repenning, A. (2010). Towards the automatic recognition of computational thinking for adaptive visual language learning. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 59-66). doi:10.1109/VLHCC.2010.17

Michotte, A. (1963). *The Perception of causality.* New York, NY: Basic Books.

Repenning, A. (2011). Making programming more conversational. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 191-194). doi:10.1109/VLHCC.2011.6070398

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). New York, NY: ACM.

Squire, K. (2003). Video games in education. *International Journal of Intelligent Games & Simulation*, *2*(1), 49-62.

Stolee, K. T., & Fristoe, T. (2011, March). Expressing computer science concepts through Kodu game lab. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 99-104). New York, NY: ACM.

Wieman, C. E., Adams, W. K., & Perkins, K. K. (2008). PhET: Simulations that enhance learning. *Science*, *322*(5902), 682-683.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *366*(1881), 3717-3725.