# Principles of Computational Thinking Tools

**Alexander Repenning, Ashok R. Basawapatna, and Nora A. Escherle**

**Abstract** Computational Thinking is a fundamental skill for the twenty-first century workforce. This broad target audience, including teachers and students with no programming experience, necessitates a shift in perspective toward Computational Thinking Tools that not only provide highly accessible programming environments but explicitly support the Computational Thinking Process. This evolution is crucial if Computational Thinking Tools are to be relevant to a wide range of school disciplines including STEM, art, music, and language learning. Computational Thinking Tools must help users through three fundamental stages of Computational Thinking: problem formulation, solution expression, and execution/evaluation. This chapter outlines three principles, and employs AgentCubes online as an example, on how a Computational Thinking Tool provides support for these stages by unifying human abilities with computer affordances.

**Keywords** Computational Thinking Process • Three stages of the Computational Thinking Process • Computational Thinking Tools • Principles of Computational Thinking Tools

## Introduction

The term Computational Thinking (CT), popularized by Jeannette M. Wing (2006), had previously been employed by Papert (1996) in the inaugural issue of Mathematics Education. Papert considered the goal of CT to *forge explicative ideas* through the use of computers. Employing computing, he argued, could result in ideas that are

A. Repenning (✉) • N.A. Escherle
School of Education, University of Applied Sciences and Arts Northwestern Switzerland FHNW, Windisch 5210, Switzerland
e-mail: alexander.repenning@fhnw.ch; nora.escherle@fhnw.ch

A.R. Basawapatna
Department of Mathematics and Computer Information Systems, SUNY Old Westbury, Old Westbury, NY 11568, USA
e-mail: basawapatnaa@oldwestbury.edu



P.J. Rich, C.B. Hodges (eds.), *Emerging Research, Practice, and Policy on Computational Thinking*, Educational Communications and Technology: Issues and Innovations, DOI 10.1007/978-3-319-52691-1_18

more accessible and powerful. Meanwhile, numerous papers, e.g., Grover and Pea (2013), and reports, e.g., National Research Council (2010), have created many different definitions of CT. Recently, Wing (2014) followed up her seminal call for action paper with a concise operational definition of CT: "Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out."

While the term Computational Thinking is relatively new, the process implied by Wing can be recognized as a computationally enhanced version of the well-established scientific method. Based on Wing's definition, the Computational Thinking Process (Fig. 1) can be segmented into three stages. The example in Fig. 1 of a mudslide simulation is used to illustrate the three Computational Thinking Process stages:

(1) Problem formulation (abstraction): Problem formulation attempts to conceptualize a problem verbally, e.g., by trying to formulate a question such as "How does a mudslide work?" or through visual (Arnheim, 1969) thinking, e.g., by drawing a diagram identifying objects and relationships.
(2) Solution expression (automation): The solution needs to be expressed in a non-ambiguous way so that the computer can carry it out. Computer programming enables this expression. The rule in Fig. 1 expresses a simple model of gravity: if there is nothing below a mud particle, it will drop down.

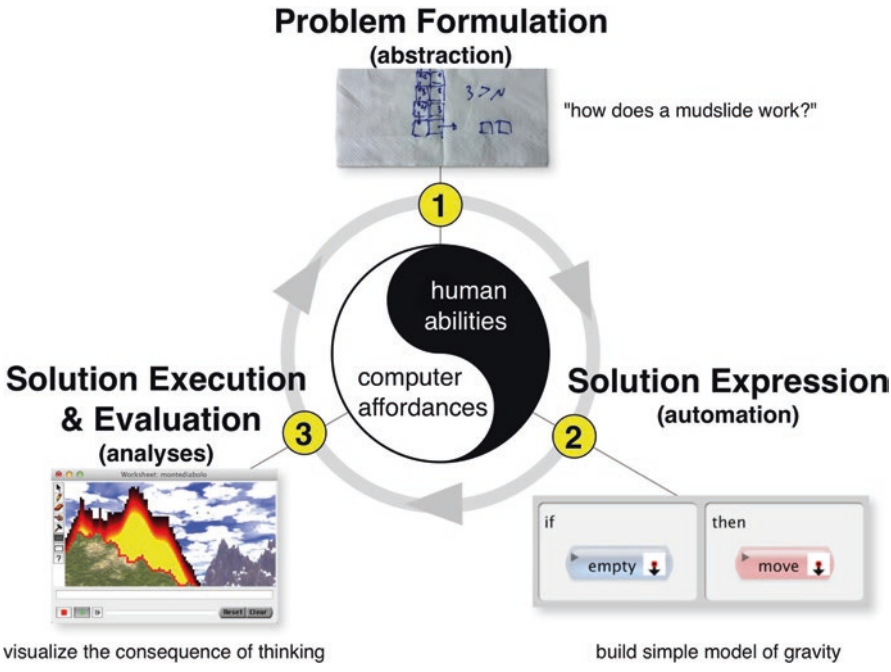


**Fig. 1** Three stages of the Computational Thinking Process

(3) Execution and evaluation (analysis): The computer executes the solution in ways that show the direct consequences of one's own thinking. Visualizations—for instance, the representation of pressure values in the mudslide as colors—support the evaluation of solutions.

As shown in Fig. 1, Computational Thinking is an iterative process describing *thinking with computers* by synthesizing human abilities with computer affordances. The three stages describe different degrees of human and computer responsibilities. The solution execution appears to be largely the responsibility of the computer and the problem expression largely the responsibility of the human. Although problem formulation is typically considered the responsibility of the human, computers can help support the conceptualization process as well, for instance, through facilitating visual thinking.

## Principles of Computational Thinking Tools

The fundamental goal of a Computational Thinking Tool is to support all stages of the Computational Thinking Process outlined above. Programming should be, and can be, an exciting new literacy in the sense described by diSessa (2000) enabling constructivist learning for all (see Yager, 1995). Using traditional programming languages severely limits this practice outside of computer science class contexts. For example, a student in a STEM class attempting to make a basic predator prey simulation with traditional programming languages may have to write hundreds of lines of code. Conversely, the goal of Computational Thinking Tools leads to three core principles corresponding to the three stages of the CT process. Computational Thinking Tools should support:

(1) Problem formulation: Similar to playing with numbers in a spreadsheet, using a mind map tool, or just doodling on a whiteboard, Computational Thinking Tools should empower users to explore representations without the need to code.
(2) Solution expression: Computational Thinking Tools should employ end-user programming approaches (see Lieberman, Paternò, & Wulf, 2006; Repenning, 2001), to allow computer users who may not have or may not want to gain professional programming experience and to create relevant computational artifacts such as games (Repenning et al., 2015) and simulations (Repenning, 2001).
(3) Solution execution and evaluation: Computational Thinking Tools should include accessible execution visualization mechanisms helping users to comprehend and validate computational artifacts such as simulations.

The AgentCubes online end-user programming environment (Ioannidou, Repenning, & Webb, 2009; Repenning, 2013b; Repenning & Ioannidou, 2006; Repenning et al., 2014) will be employed as an example to illustrate these principles,

but these principles can be applied to any CT Tool. The AgentCubes user interface is relatively simple. The toolbar at the top of Fig. 3 provides a number of controls to start/stop a simulation, to manage worlds, and to operate the 3D camera. The panel to the left contains all the user-defined agents. The top panel is the current world. The three bottom panels contain the drag and drop programming environment with the condition palette to the left, the agent behavior in the middle, and the palette of actions to the right. The following sections discuss the three Computational Thinking Tool principles and provide concrete examples through AgentCubes.

## *Supporting Problem Formulation*

Problem formulation is a conceptualization process (Repenning et al., 2015) dealing with abstractions often based on verbal or visual thinking, which can be supported by tools. Computational Thinking Tools can support visual thinking by offering various evocative spatial metaphors. Mind map tools capture concepts as nodes and links (Willis & Miertschin, 2005). Spreadsheets (B. A. Nardi & Miller, 1990) are two-dimensional grids containing numbers and strings. The versatile nature of grids has helped spreadsheets to become the world's most used programming tools. Tools such as Boxer (diSessa, 1991) and ToonTalk (Kahn, 1996) employ the notion of microworlds based on containers to represent relationships. In logo, Papert (1993) argues the notion of a turtle helps users comprehend difficult geometric transformations through body syntonicity, that is, the ability for people to project themselves, as turtle, into geometric microworlds. Papert (1993) and Turkle (2007) consider the use of *evocative objects to think with* as a powerful conceptualization approach. All these tools help the forging of abstractions serving as the beginning of a path from problem formulation to solution expression. Wing (2008) suggests that finding these kinds of abstractions is an essential part of Computational Thinking: "In working with rich abstractions, defining the 'right' abstraction is critical. The abstraction process—deciding what details we need to highlight and what details we can ignore—underlies computational thinking" (p. 3718).

Abstractions need to be made explicit to enable transfer. Ideally, Computational Thinking Tools should not only support users to find rich, evocative abstractions but also make these abstractions explicit in order to facilitate their transfer and application within other problem-solving contexts. For instance, the use of phenomenalistic (Michotte, 1963) abstractions describing object interactions such as *collision* and *diffusion* was found to support student formulation of STEM simulations in middle school curricula (Koh, Basawapatna, Bennett, & Repenning, 2010; Repenning et al., 2015). In our research, the patterns found to be especially helpful in allowing students to create elements of games and simulations we termed Computational Thinking Patterns (CTPs). Figure 2 lists examples of Computational Thinking Patterns. For example, the *collision* CTP describes both the interaction of a truck hitting a frog in a Frogger-like game and the interaction of molecules colliding in a STEM simulation. Similarly the *generate* CTP could describe a ship shooting lasers

- **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.

**Fig. 2** Examples of Computational Thinking Patterns

in a Space Invaders-type game but also two foxes mating and creating offspring in a predator prey simulation. Learning these CTPs provides students with a useful high-level language to begin thinking about a problem before coding begins, and previous research has shown that novice users can recognize these patterns across contexts and implement them in their project creations (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011).

### How AgentCubes Supports Problem Formulation

AgentCubes online supports the problem formulation stage similarly to a mind map tool by enabling users to organize information visually setting the stage for coding. At the problem formulation stage, AgentCubes online can be used much like a whiteboard is used for drawing. The 2D or 3D objects, called agents, created by users are similar to Papert's objects to think with (Papert, 1993). In AgentCubes, information can be organized in 3D space to create 3D worlds. Similar to Minecraft, users create one-, two-, or three-dimensional grids and stacks by placing agents using the pen tool (Repenning et al., 2014). At this stage no coding is necessary. Users can explore their worlds by employing 3D camera tools to navigate or manipulate their worlds by adding, removing, and rearranging agents. AgentCubes allows users to select any agent and assume its perspective by switching to first-person camera mode. This ability, we speculate, may help to achieve the body syntonicity (Repenning & Ioannidou, 2006) that Papert is referring to.

Visual thinking is supported by AgentCubes online through a four-dimensional grid structure called the agent matrix (Fig. 3). The grid is based on cells organized as rows, columns, and layers. Each cell, in turn, contains a stack of agents. Agents can be simple textured shapes such as cubes, spheres, or cylinders but can also be quite sophisticated user-created 3D shapes implemented as inflatable icons (Repenning et al., 2014). Users' ability to produce their own 3D shapes has been identified as an important creativity tool to overcome affective challenges of programming, but it can also be useful to quickly sketch out 3D worlds similar to the use of a cocktail napkin in the formulation stage depicted in Fig. 1.

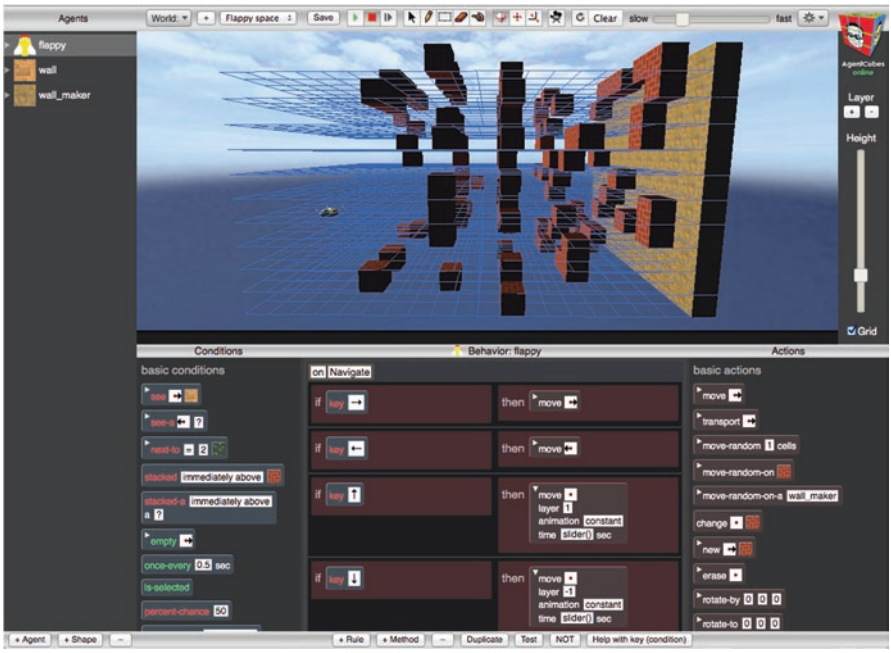


**Fig. 3** AgentCubes online environment depicting a side view of an example "Flabby Bird 3D" game

As an example, an interesting problem could be how to generalize a 2D side-scrolling game into a 3D scrolling game. In Fig. 3, the grid has been enabled to show the AgentCubes cell structure of a game called "Flabby Bird 3D," which is a generalization of the popular 2D scrolling phone game "Flappy Bird." In Flabby Bird 3D, the objective is to navigate a bird called Flabby past oncoming cubes. Usually, a player would see this game from the first-person camera viewpoint of Flabby (Fig. 4). The enabled grid helps to reveal the 3D scrolling approach of the game. To the right there is a solid wall of cube maker agents creating cubes (an example of the *generate* CTP) with an increasing probability depending on the level of the game. These cubes are flying toward Flabby as depicted in Fig. 4. Playing the game, by seeing it from Flabby's point of view, the player gets the illusion of flying through a never-ending labyrinth of walls. To make the game more challenging, the approaching walls reconfigure occasionally.

Breaking down game descriptions into explicit abstractions enables students to transition from problem formulation to solution expression (Repenning et al., 2015). Computational Thinking Patterns serve as framework of useful abstractions describing the interactions between objects and the interaction of users with objects. For instance, the creation of Flabby Bird involves the implementation of various Computational Thinking Patterns such as *collision*, *generation*, *absorption*, and *keyboard control*. Part of the support structure for this activity is external. For instance, some teachers hang up posters describing the Computational Thinking

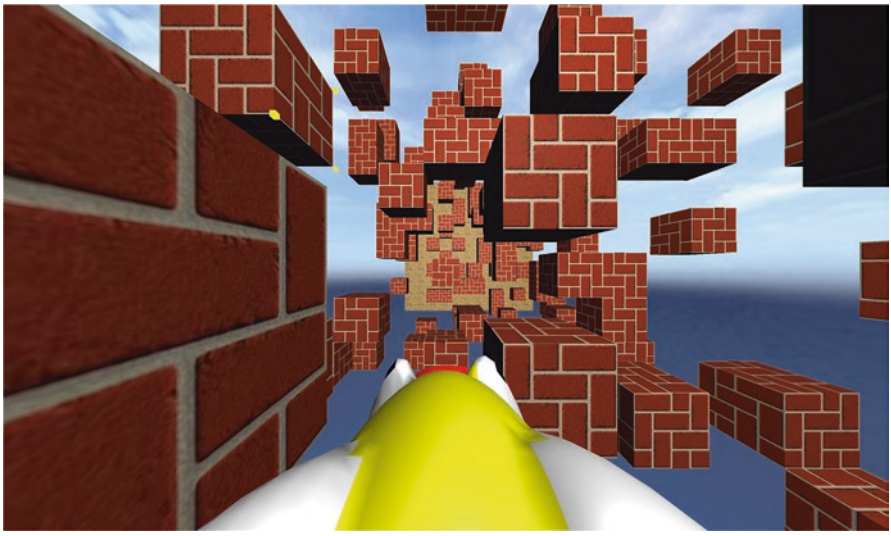

**Fig. 4** First-person view perspective of the "Flabby Bird 3D" game

Patterns and make students refer to these posters when working on problem formulation tasks. However, it is essential that the Computational Thinking Tool provides for a solution expression that is an intuitive implementation of the problem formulation. It should be noted that this step can be fully integrated into the tool itself. For example, tools that allow users to program agents directly, through Computational Thinking Patterns, have successfully been piloted in the past, further bridging the act of problem formulation with solution expression (Basawapatna, Repenning, & Lewis, 2013).

## *Supporting Solution Expression*

The goal of Computational Thinking is to be an instrument for problem solving that is not limited to computer scientists or professional programmers. For example, assuming an educational context, such as STEM classes, Computational Thinking Tools need to be viable in noncomputer science classes by avoiding the need for difficult and excessive coding. CT employing traditional programming tools is likely to introduce a significant amount of accidental complexity, as opposed to dealing with the intrinsic complexity (Dijkstra, 2001) of the problem-solving process. If the intended outcome is to become a professional programmer, then this approach may be highly effective or indeed entirely necessary.

If instead the goal is to become a Computational Thinker, then the resulting overhead and lack of support may turn into an insurmountable educational obstacle. Focusing less on the notion of essence but on understandable mappings, natural

programming (Myers, Pane, & Ko, 2004) attempts to better align the expression of a solution with the problem formulation based on peoples' intuitive comprehension of semantics such as the use of Boolean operators. Rittel differentiated the notion of human computer interaction from human problem-domain interaction (Rittel & Webber, 1984) to clarify this important philosophical dichotomy. Guzdial (2015) reached a similar conclusion in the context of computing education by suggesting that "If you want students to use programming to learn something else [e.g., how to author a simulation] then limit how much programming you use" (p. 48). The limitation of accidental complexity can be supported at three different levels:

(1) Syntax: Visual programing approaches such as drag and drop programming (Conway et al., 2000; Repenning & Ambach, 1996; Resnick et al., 2009a; 2009b) can avoid frustrating syntactic challenges such as missing semicolons.
(2) Semantics: Live programming (Burckhardt et al., 2013; McDirmid, 2013; McDirmid, 2007) and similar approaches help users to understand the meaning of programs by illustrating the consequences of changes to programs.
(3) Pragmatics: Domain-oriented (Fischer, 1994) or task-specific (B. Nardi, 1993) programming languages support users in employing programming languages to achieve their goals.

### How AgentCubes Supports Solution Expression

At the syntactic level, AgentCubes offers drag and drop programming, which its predecessor, AgentSheets, pioneered over 20 years ago (Repenning & Ambach, 1996). A first version of AgentSheets initially introduced the idea of agent-based graphical rewrite rules (Repenning, 1994, 1995), a programming by example (Repenning & Perrone, 2000; Repenning & Perrone-Smith, 2001) approach to define the behavior of agents by demonstrating it. However, the graphical rewrite rules were ultimately considered to be too constraining (Schneider & Repenning, 1995). Meanwhile, the benefits of drag and drop programming have become quite clear, and consequently drag and drop programming has proliferated to a very large number of programming environments for kids (Conway et al., 2000; Resnick et al., 2009a; 2009b).

Semantic support is considerably harder than syntactic support (Repenning, 2013a). At the level of semantics, AgentCubes offers not only live programming (McDirmid, 2013; McDirmid, 2007) but also a technique called Conversational Programming (Repenning, 2013a). Conversational Programming will observe the agent a user is interested in and then annotate the program behaviors of that particular agent in its particular situation by running the program one step into the future to illustrate which agent behavior rules will evaluate to true, which will evaluate to false, and which rules will not be tested. For instance, in a Frogger-like game, a user can click the frog agent and look at its behavior rules to see what will happen to the frog after it has just jumped in front of a car moving toward it. In this case, if the frog-car *collision* pattern is programmed correctly, the behavior rule wherein the frog dies and the game restarts will be annotated by Conversational Programming to appear as true.


**Fig. 5** 15-square puzzle

Of the three levels, pragmatics is the most challenging one to support. One might naturally want to have a simple mapping between the problem domain and the solution domain. However, the least amount of code cannot be the only objective. For example, languages such as APL are well known for their parsimonious nature but not for their general readability. Instead, programs should be short and intuitive expressions of a given idea. An example may help to illustrate this.

The 15-square puzzle, shown in Fig. 5, is a classic children's toy. The game consists of sliding 15 numbered squares into a sorted arrangement, 1–15, in a $4 \times 4$ grid. Many computer program implementations of the game exist. From a CT point of view, the core idea is simple: click the square you want to slide into the empty space. From a coding point of view, however, efforts can vary widely. A Python program to implement the "click to slide" functionality (see Sweigart, 2010) quickly runs into hundreds of lines of code not including the functionality to solve the puzzle. The view here is not to be negative regarding coding. If a CS class codes the 15-square puzzle to learn about arrays, loops, animations, or Python syntax, then writing the 300 lines of code could be extremely beneficial.

An AgentCubes implementation, in contrast, will include very little coding overhead. The "click to slide" functionality requires only four rules, checking if there is an empty spot adjacent to the clicked square and then moving into that spot. This is depicted in Fig. 6. Trading in clarity for brevity, one could even employ the more arcane MoveRandomOn (background) AgentCubes action to solve the 15-square puzzle benchmark in a single line of code. Comparing Python to AgentCubes seems hardly fair. In AgentCubes the notion of a grid, animations, and even numbered squares already exists. This is what domain orientation (Fischer, 1994) can do. It reduces coding overhead by providing and implementing abstractions to help users express a solution succinctly.

Of course, domain orientation introduces trade-offs. For instance, it would not be advisable to write a compiler in AgentCubes. Similar to spreadsheets—which have been used creatively to create amazing projects such as flight simulators and

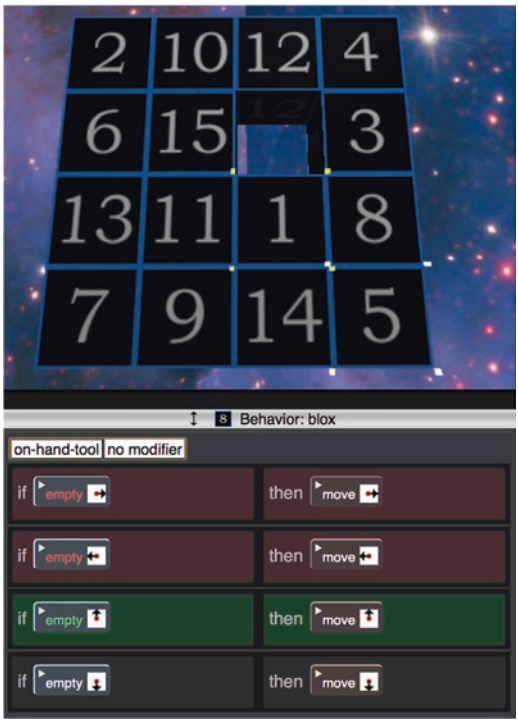


**Fig. 6** AgentCubes online implementation of the 15-square puzzle with four rules

planetary models—AgentCubes' grid structure maps well onto a wide variety of projects such as 2D/3D games, simulations, and cellular automata. For instance, a simple version of the Pac-Man game can be implemented in just ten rules (if/then statements) including collaborative AI (Repenning, 2006) and win/lose detection.

Studies show that students can use such system affordances of AgentSheets and AgentCubes to successfully implement the Computational Thinking Patterns planned in the formulation step in game and simulation development (Repenning et al., 2015). Studies also show that users are highly motivated to create these artifacts, speaking to the power of reducing coding overhead (Repenning, Basawapatna, Assaf, Maiello, & Escherle, 2016). Guzdial (2008) points out the importance of avoiding coding overhead in education and refers to a number of languages explored in computer science education to establish essence by employing implicit loops and other task-specific (B. Nardi, 1993) constructs. An example of this approach in AgentCubes is the built-in management of parallelism. For instance, even a very large number of agents, looking like boxes, moving around randomly in a three-dimensional world, will automatically reshuffle and stack up correctly in parallel with very little code. Computing trajectories that can be executed in parallel, determining the order of boxes stacked up, would be complex code to write.

### *Supporting Execution and Evaluation*

The execution and evaluation stage can be supported by helping users debug their programs as well as reveal their misconceptions. Pea (1983) describes debugging as "systematic efforts to eliminate discrepancies between the intended outcomes of a program and those brought about through the current version of the program" (p. 3). Given that the computer does not currently "understand" the problem, it will not be able to automatically compute these discrepancies, but there are still strategies for Computational Thinking Tools to aid the debugging process. One strategy is to simply reduce the gap between solution expression and solution execution and evaluation. Punch cards are the classical negative example resulting in an extremely large gap. As this gap increases, users quickly lose sight of the causal relation between changes made to a program and manifestations of different behaviors exhibited by running the modified program (Repenning, 2013a).

Live programming (McDirmid, 2013) can help by enabling users to instantly see the consequences of any change to a program. Unfortunately, there are issues such as the halting problem in computer science theory with practical consequences, suggesting that it is not actually possible to determine all consequences of arbitrary program changes. However, for a more constrained class of programs, including spreadsheets, this is not a problem. Very much in the spirit of live programming, spreadsheets will instantly update results when formulae or cell values are changed by a user.

A Computational Thinking Tool would support visualization through the inclusion of easy-to-use visualization affordances. Additionally, a Computational Thinking Tool may apply the idea of visualization to itself by annotating programs in ways to make discrepancies between the programs users have and the ones they want more understandable (Repenning, 2013a).

#### How AgentCubes Supports Execution and Evaluation

To support the goal of visualizing the consequences of one's own thinking, a number of visualization techniques are included in AgentCubes. In the mudslide example (Fig. 1 solution execution and evaluation), it helps considerably to understand the pressure distribution among the thousands of agents employed in the model. The simple visualization scheme mapping each pressure value into a single color helps the forging of explicative ideas by depicting pressure buildup.

Particularly useful when making simulations, AgentCubes supports the plotting of simulation properties. An example would be to plot the number of predators and prey in an ecological simulation. One can also use 3D plotting to visualize value fields in real time. For instance, in a city traffic simulation, 3D plots (Fig. 7) show the spatial distribution of wait times in the city over the world grid itself. Finally, AgentCubes online narrows the gap between solution expression and execution through Conversational Programming (introduced above in Supporting Solution Expression) (Repenning, 2013a), extending the notion of live programming (McDirmid, 2013).

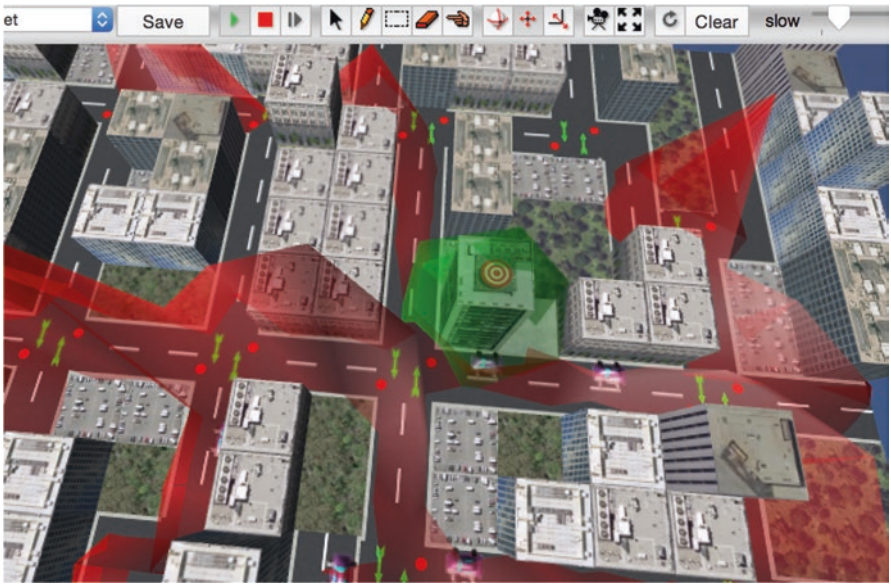


**Fig. 7** Bird's eye view of a city traffic simulation in AgentCubes with an overlaid 3D plot of traffic wait times with the higher red peaks indicating a longer wait

Even when a game is not running, by selecting an agent in the world, AgentCubes will execute relevant code fragments one step into the future and annotate the code, specifying which rule will execute, in order to visualize potential discrepancies between the programs users have and the programs users want (Pea, 1983). This indicator can guide users into another iteration cycle depicted in Fig. 1 yielding more useful representations.

## Conclusions

Computational Thinking Tools should support Papert's vision of enabling users to forge explicative ideas through the use computers. By minimizing coding overhead, Computational Thinking Tools can allow all users to focus on the essence of abstraction, automation, and analysis. In contrast to traditional programming environments, Computational Thinking Tools support all three stages, problem formulation, solution expression, and execution and evaluation, of the Computational Thinking Process. This support will make Computational Thinking feasible to a wide range of applications including STEM, art, music, and language.

**Acknowledgments** This work is supported by the Hasler Foundation and the National Science Foundation under Grant Numbers 0833612, 1345523, and 0848962. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these foundations.

## References


Arnheim, R. (1969). *Visual thinking*. Berkley, CA: University of California Press.

A. Basawapatna, K. H. Koh, A. Repenning, D. C. Webb, & K. S. Marshall. (2011). *Recognizing computational thinking patterns*. Paper presented at the the 42nd ACM technical symposium on computer science education (SIGCSE), Dallas, TX, USA.

Basawapatna, A. R., Repenning, A., & Lewis, C. H. (2013). *The simulation creation toolkit: An initial exploration into making programming accessible while preserving computational thinking*. Paper presented at the 44th ACM technical symposium on computer science education (SIGCSE 2013), Denver, CO, USA.

Burckhardt, S., Fahndrich, M., Halleux, P. D., McDirmid, S., Moskal, M., Tillmann, N., & Kato, J. (2013). *It's alive! continuous feedback in UI programming*. Paper presented at the proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation, Seattle, WA, USA.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., & Pausch, R. (2000). *Alice: Lessons learned from building a 3D system for novices*. Paper presented at the CHI 2000 conference on human factors in computing systems, The Hague, Netherlands.

Dijkstra, E. W. (2001). The end of computing science? *Communications of the ACM, 44*(3), 92. doi:10.1145/365181.365217.

diSessa, A. A. (1991). An overview of boxer. *Journal of Mathematical Behavior, 10*, 3–15.

diSessa, A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT.

Fischer, G. (1994). Domain-oriented design environments. In *Automated software engineering* (Vol. 1, pp. 177–203). Boston, MA: Kluwer Academic.

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43. doi:10.3102/0013189X12463051.

Guzdial, M. (2008). Education: Paving the way for computational thinking. *Communications of the ACM, 51*, 25–27.

Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics, 8*, 1.

Ioannidou, A., Repenning, A., & Webb, D. (2009). AgentCubes: Incremental 3D end-user development. *Journal of Visual Language and Computing, 20*(4), 236–251.

Kahn, K. (1996). *Seeing systolic computations in a video game world.* Paper presented at the proceedings of the 1996 IEEE symposium of visual languages, Boulder, CO, USA.

Koh, K. H., Basawapatna, A., Bennett, V., & Repenning, A. (2010). *Towards the automatic recognition of computational thinking for adaptive visual language learning*. Paper presented at the conference on visual languages and human centric computing (VL/HCC 2010), Madrid, Spain.

Lieberman, H., Paternò, F., & Wulf, V. (Eds.). (2006). *End user development* (Vol. 9). Dordrecht: Springer.

McDirmid, S. (2007). *Living it up with a live programming language.* Paper presented at the proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems and applications (OOPSLA '07).

McDirmid, S. (2013). *Usable live programming*. Paper presented at the SPLASH onward!, Indianapolis, IN, USA.

Michotte, A. (1963). The perception of causality. (T. R. Miles, Trans.). London: Methuen

Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural programming languages and environments. *Communications of the ACM, 47*(9), 47–52. doi:10.1145/1015864.1015888.

Nardi, B. (1993). *A small matter of programming*. Cambridge, MA: MIT.

Nardi, B. A., & Miller, J. R. (1990). *The spreadsheet interface: A basis for end user programming*. Paper presented at the INTERACT 90–3rd IFIP international conference on human-computer interaction, Cambridge, http://www.miramontes.com/writing/spreadsheet-eup/

National Research Council, Committee for the Workshops on Computational Thinking, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences.

(2010). Report of a workshop on the scope and nature of computational thinking. Washington, DC: National Academies.

Papert, S. (1993). *The children's machine*. New York, NY: Basic Books.

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning, 1*(1), 95–123.

Pea, R. (1983). *LOGO programming and problem solving*. Paper presented at symposium of the annual meeting of the American Educational Research Association (AERA), "Chameleon in the Classroom: Developing Roles for Computers" Montreal, Canada, April 1983.

Repenning, A. (1994). *Bending icons: Syntactic and semantic transformation of icons*. Paper presented at the proceedings of the 1994 IEEE symposium on visual languages, St. Louis, MO.

Repenning, A. (1995). *Bending the rules: Steps toward semantically enriched graphical rewrite rules*. Paper presented at the proceedings of visual languages, Darmstadt, Germany.

Repenning, A. (2001). *End-user programmable simulations in education*. Paper presented at the HCI international 2001, New Orleans.

Repenning, A. (2006). *Collaborative diffusion: Programming antiobjects*. Paper presented at the OOPSLA 2006, ACM SIGPLAN international conference on object-oriented programming systems, languages, and applications, Portland, Oregon.

Repenning, A. (2013a). *Conversational programming: Exploring interactive program analysis*. Paper presented at the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming and software (SPLASH/Onward! 13), Indianapolis, Indiana, USA.

Repenning, A. (2013b). Making programming accessible and exciting. *IEEE Computer, 18*(13), 78–81.

Repenning, A., & Ambach, J. (1996). *Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing*. Paper presented at the 1996 IEEE symposium of visual languages, Boulder, CO.

Repenning, A., & Ioannidou, A. (2006). *AgentCubes: Raising the ceiling of end-user development in education through incremental 3D*. Paper presented at the IEEE symposium on visual languages and human-centric computing 2006, Brighton, UK.

Repenning, A., & Perrone, C. (2000). Programming by analogous examples. *Communications of the ACM, 43*(3), 90–97.

Repenning, A., & Perrone-Smith, C. (2001). Programming by analogous examples. In H. Lieberman (Ed.), *Your wish is my command: Programming by example* (Vol. 43, pp. 90–97). San Francisco, CA: Morgan Kaufmann Publishers.

Repenning, A., Basawapatna, A., Assaf, D., Maiello, C., & Escherle, N. (2016). *Retention of flow: Evaluating a computer science education week activity*. Paper presented at the special interest group of computer science education (SIGCSE 2016), Memphis, Tennessee.

Repenning, A., Webb, D. C., Brand, C., Gluck, F., Grover, R., Miller, S., et al. (2014). Beyond minecraft: Facilitating computational thinking through modeling and programming in 3D. *IEEE Computer Graphics and Applications, 34*(3), 68–71. doi:10.1109/MCG.2014.46.

Repenning, A., Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., et al. (2015). Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *Transactions on Computing Education (TOCE), 15*(2), 1–31. doi:10.1145/2700517.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., & Kafai, Y. (2009a). Scratch: Programming for all. *Communincation of the ACM, 52*(11), 60–67.

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

Rittel, H., & Webber, M. M. (1984). Planning problems are wicked problems. In N. Cross (Ed.), *Developments in design methodology* (pp. 135–144). New York, NY: Wiley.

Schneider, K., & Repenning, A. (1995). *Deceived by ease of use: Using paradigmatic applications to build visual design.* Paper presented at the proceedings of the 1995 symposium on designing interactive systems, Ann Arbor, MI.

Sweigart, A. (2010). *Invent your own computer games with Python, A beginner's guide to computer programming in Python*.
Turkle, S. (2007). *Evocative objects: Things we think with*. Cambridge, MA: MIT.
Willis, C. L., & Miertschin, S. L. (2005). *Mind tools for enhancing thinking and learning skills*. Paper presented at the proceedings of the 6th conference on information technology education, Newark, NJ, USA.
Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35.
Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society, 2008*(366), 3717–3725.
Wing, J. M. (2014). *Computational thinking benefits society*. http://socialissues.cs.toronto.edu/index.html%3Fp=279.html
Yager, R. (Ed.). (1995). *Constructivism and learning science*. Mahway, NJ: Lawrence Erlbaum Associates.