Computational Thinking Tools

Alexander Repenning[§^]
[§]University of Colorado
Boulder, Colorado, 80309, USA
[^]School of Education, FHNW
5200 Brugg-Windisch, Switzerland

Ashok Basawapatna*, Nora Escherle^
*Dept. of Math and Computer Information Systems
State University of New York College at Old Westbury
Old Westbury, NY

Abstract–Computational Thinking is an essential skill for all students in the 21st Century. A fundamental question is how can we create computer affordances to empower novice teachers and students, in a variety of STEM and art disciplines, to think computationally while avoiding difficult overhead emerging from traditional coding? Over the last 20 years we have iteratively developed tools that aim to support computational thinking. As these tools evolved a philosophy emerged to support Computational Thinking by joining human abilities with computer affordances. Chief among these findings is that supporting Computational Thinking is much more than making coding accessible. Computational Thinking Tools aim to minimize coding overhead by supporting users through three fundamental stages of the Computational Thinking development cycle: problem formulation, solution expression, and solution execution/evaluation.

Keywords—computational thinking tools; end-user programming; K-12 education; computational thinking

I. Introduction

Few question the potential value of Computational Thinking (CT) in schools [1] but it is less clear how traditional coding tools can be successfully employed for Computer Science Education without introducing difficult overhead. Particularly in subject areas where coding is not the focus, e.g. STEM, language, arts, and music, coding overhead quickly turns into an insurmountable educational obstacle. No matter how educational, a middle school teacher teaching predator prey interactions in biology is not likely to engage in CT if programming a simple model results in the need to write hundreds or even thousands of lines of code.

The goal of schools to produce Computational Thinkers, as opposed to coders, requires a shift in the methods and tools employed by teachers. Starting with AgentSheets [2], we have built, and gradually evolved, a number of programming tools aimed specifically at Computer Science Education over the last 20 years. Moving beyond tools to the development of curricula such as Scalable Game Design [3], assessing Computational Thinking and conducting teacher professional development worldwide [4, 5] the notion of Computational Thinking Tools gradually emerged to capture the educational needs in Computer Science education. This paper outlines a philosophy behind the Computational Thinking process and, suggests how Computational Thinking Tools not only support this process but, at the same time, keep coding overhead minimal.

978-1-5090-0252-8/16/$31.00 ©2016 IEEE

II. The Computational Thinking Process

The term Computational Thinking (CT), popularized by Wing [1], had previously been employed by Papert in the inaugural issue of Mathematics Education [6]. 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 more accessible and powerful. Meanwhile, numerous papers [7] and reports have created many different definitions of CT. Recently, Wing followed up her seminal call for action paper with a concise operational definition of CT [8]:

> "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 (Figure 1) can be segmented into three stages. Exploring the notion of computational tools to support Computational Thinking, this paper focuses on the execution of solutions by machines and largely omits approaches dealing with human execution of solutions such as unplugged activities [9]. The example in Figure 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 [10] 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 Figure 1 expresses a simple model of gravity: if there is nothing below a mud particle it will drop down.
3. ***Execution & Evaluation (Analysis)***. The solution gets executed by the computer 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.

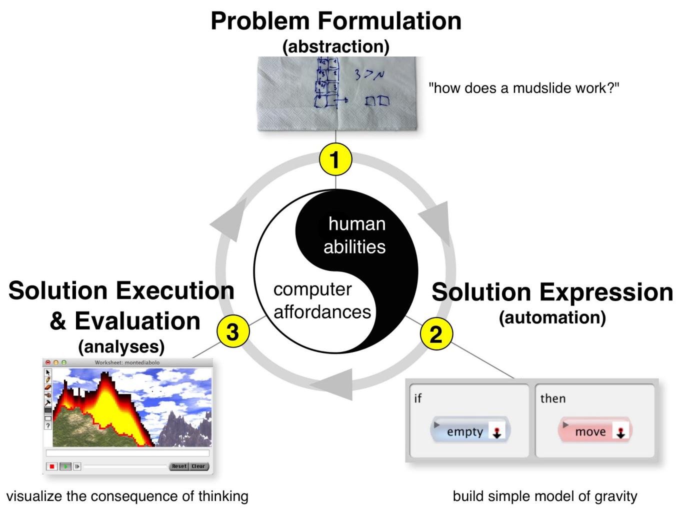


Fig. 1. The Computational Thinking Process

As shown in Figure 1, Computational Thinking is an iterative process describing *thinking with computers* by synthesizing human abilities with computer affordances. The three stages describe different ratios 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 typically considered the responsibility of humans, problem formulation is perhaps more of a mix. Computers can help support the conceptualization process as well, for instance, through facilitating visual thinking.

The iterative nature of Computational Thinking typically suggests many iterations through the three stages. The boundary between the problem formulation and the solution expression stage is sometimes quite fuzzy because the solution approaches known to humans may strongly flavor their personal conceptualizations at the problem formulation stage. Similarly, the boundary between problem expression and problem execution may become more fluid with modern programming approaches such as live programming [11] which executes programs as they are being created.

The vision for Computational Thinking Tools is to *support and integrate the three stages of the Computational Thinking Process*. Certainly, any kind of programming tool can be employed for computational thinking. End-user programming tools, for instance, are focused on the support of the solution expression by making programming more accessible. However, Computational Thinking Tools should go further by providing additional support for the problem formulation as well as the problem execution & evaluation stages of the Computational Thinking process. We will now review these three stages in more detail. Though there are many examples of tools that enable Computational Thinking, we will present brief examples from our research including AgentCubes online [12] and the Scalable Game Design Project [3].

III. SUPPORTING THE COMPUTATIONAL THINKING PROCESS

The three stages of the Computational Thinking Process should be supported with Computational Thinking Tools. These tools may be computational in nature but do not have to be. For instance, a cocktail napkin that is used for conceptual doodling is a simplistic tool facilitating problem formulation. The following three sections give examples of various Computational Thinking process support mechanisms.

A. *Supporting Problem Formulation*

Problem formulation is a conceptualization process [3] often based on verbal or visual thinking, which can be supported by a great variety of tools, ranging from informal drawings on napkins to arranging objects within spreadsheet-like programming environments. For problem formulation concepts developed to become powerful, transferable abstractions they need to be scaffolded [13] explicitly.

1) *Conceptualization through verbal and visual thinking*

The use of verbal thought goes a long way to support problem formulation. Framing a problem as narrative can help users comprehend relationships between objects. In object-oriented design, for instance, the identification of nouns and verbs in a problem description supports the conceptualization of software components such as classes and methods. Experience from the Scalable Game Design project [3] indicates that capturing the core idea of a game to be built as a story is extremely useful.

Visual thinking supports problem formulation for applications, such as simulations, where narrative approaches are sometimes less suited. Arnheim [10] suggests that visual thinking is based on processes unfolding at a level of perception and not necessarily at the level of language. In the case of the mudslide, the development of a textual narrative may not be the most practical abstraction tool. In contrast, the abstraction of a mudslide as a two dimensional array of mud chunks interacting with each other is extremely beneficial but requires scaffolding [13].

Computational Thinking Tools can support visual thinking by offering various evocative spatial metaphors. Mindmap tools capture concepts as nodes and links [14]. Spreadsheets [15] 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 [16] and ToonTalk [17] employ the notion of microworlds based on containers to represent relationships. In Logo, Papert argues, the notion of a turtle helps users comprehend difficult geometric transformations through body syntonicity [18], that is, the ability for people to project themselves, as turtle, into geometric microworlds. Papert [18] and Turkle [19] consider the use of *evocative objects to think with* as powerful conceptualization approach. All these tools help the forging of abstractions serving as the beginning of a path from problem formulation to solution expression.

In end-user programming tools, for example, the problem formulation stage is supported similar to a mind map tool, or games like Minecraft and SimCity, by enabling users to organize information visually. For example, in the AgentCubes online [12] end-user programming tool, users can place objects, called agents, in a 3-D grid structured environment. 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.

These 2D or 3D user created agents are similar in spirit to Papert's objects to think with [18].

In the mudslide example, depicted in Figure 1, a user might create a mud agent and a ground agent and then set up the simulation world by stacking multiple mud agents on top of a layer of ground agents. At this stage no programming has commenced. Users can explore their simulation world by employing 3D camera tools to navigate around in the world or by adding, removing, and rearranging the mud agents. To gain a different perspective, they can select any agent and switch to first person camera mode. The ability to assume the perspective of any object in a world, we speculate, may help to achieve the body syntonicity [20] that Papert is referring to.

2) *Supporting Transfer through explicit abstractions*

Evidence from the learning sciences strongly suggests that the forging of abstraction does not just happen. Early on Pea [21] described the problem he called the *pedagogical fantasy* as the exaggerated expectation of emerging connections between programming and higher level thinking in students:

> "This idea – that programming will provide exercise for the highest mental faculties, and that the cognitive development thus assured for programming will generalize or transfer to other content areas in the child's life – is a great hope. Many elegant analyses offer reasons for this hope, although there is an important sense in which the arguments ring like the overzealous prescriptions for studying Latin in Victorian times."

Abstractions need to be made explicit to enable transfer. For instance, the use of the Computational Thinking Patterns (CTPs) as abstractions in the Scalable Game Design curriculum [3, 22] was found to be helpful to convey ideas relevant to Computational Thinking. CTPs are phenomenalistic [23] abstractions describing object interactions such as collision and diffusion. A collision, for instance, could describe the interaction of a frog and truck in a Frogger-like game or the interaction of molecules participating in Brownian motion in a STEM simulation. Students are initially exposed to CTPs in game design and then, later, implement these CTPs in subsequent games and simulations. CTPs could be considered a powerful set of abstractions serving as bridge between problem formulation and solution expression. Wing confirms that finding suitable abstractions is a critical part of computational thinking [24].

> "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."

We do not claim Computational Thinking Patterns to be an exhaustive collection of CT abstractions but just to be a useful set of abstractions relevant to CT that has been validated in the context of Scalable Game Design with a very large study [3]. Furthermore, the Computational Thinking Pattern Analyses tool [3] automatically extracts these kinds of patterns from game and simulation projects using Latent-semantics inspired pattern matching approaches. Validation also provides evidence that students can recognize and use these patterns to build projects. Some teachers have even highlighted the use of these patterns as the main reason they were able to teach middle school students succesfully in a short time and indicated that the patterns helped transcend beyond game design to STEM simulation creation [25].

B. *Supporting Solution Expression*

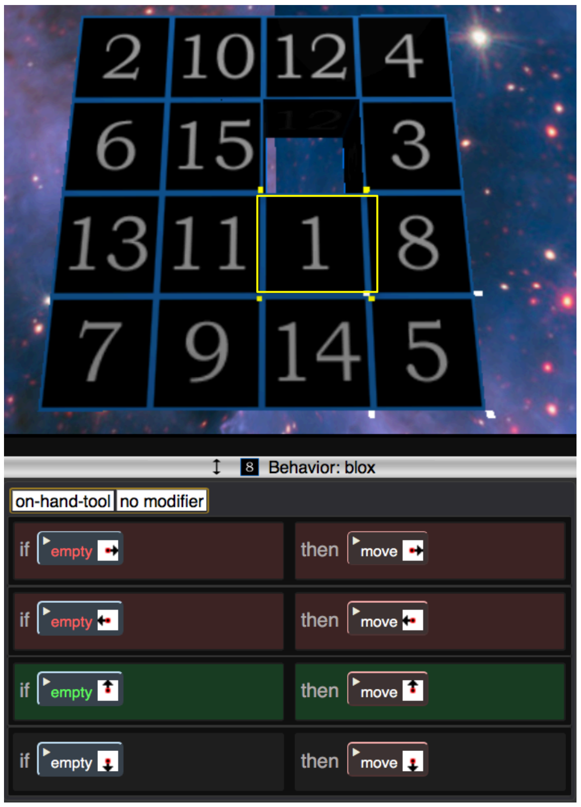
Given that even low cost computers these days provide multi core, multi Gigahertz performance, users have the right to expect more than an "empty window with a blinking cursor in the upper left corner" type of solution expression support from Computational Thinking Tools. The visual programming community has explored approaches to make programming more accessible. Approaches such as drag and drop programming [2, 26, 27] have addressed, by and large, syntactic challenges. The grand challenges of end-user programming are shifting from syntactic to semantic and pragmatic concerns. In this realm, research is less worried with the representation of code, e.g., if an action is represented as text or puzzle piece [28], but more concerned with its meaning in the context of particular situations.

Also important for Computational Thinking Tools is the need to minimize accidental complexity. Brooks [29] differentiates between the essential or intrinsic complexity, that is the complexity that is directly found or implied by the problem space, and the accidental complexity found in the solution space. Accidental complexity can be the result of employing an approach that is a mismatch for the problem. Accidental complexity is typically identified by components in the solution space that cannot be easily traced back to the problem space. Conway's Game of Life serves as a simple example problem. A cell is dead or alive and there are three simple rules determining transitions between these two states based on the states of immediate neighbor cells. These three rules are the intrinsic complexity of this problem. A programming novice may naively hope for these three rules to turn into three IF statements. If that were the case, the accidental complexity added would be zero. Unfortunately, with most programming tools, even this seemingly trivial problem can quickly turn into a perplexingly complex and potentially frustrating programming exercise. Users are forced to deal with extrinsic complexity, that is, they may have to write support code dealing with event handling, and input/output control potentially significantly exceeding the three rules implied by the game of life.

A recent study [30] trying to understand the factors that influence young women's decisions to pursue degrees, found the top two adjectives describing women's perception of computer science as "hard" and "boring." While the need to deal with accidental complexity, resulting in the need to write elaborate programs to solve even simple problems, is certainly not the only reason people are not interested in programming, addressing affective challenges [12] is an important factor in making programming more popular.

The end-user and visual programming language community has explored a number of approaches supporting a focus on essence for some time. Domain-orientation [31] provides high level languages supporting the creation of specific applications. The pinball construction kit, for instance, allowed users to build a working pinball simulation by arranging pinball components through a drag and drop interface. Task-specific [32] languages, similarly, provide powerful building blocks enabling end-users to build applications. Guzdial [33] points to a number of languages explored in computer science education to establish essence by employing implicit loops and other constructs such as spreadsheets are well known to eliminate accidental complexity caused by what Lewis calls the pumping code [34], i.e., the code needed to interact with users. Focusing less on the notion of essence but on understandable mappings, Natural programming [35] 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.

A simple example of accidental complexity occurs when trying to program the 15 squares puzzle, shown in Figure 2. This classic children's game consists of sliding 15 numbered squares into a sorted arrangement, 1-15, in a 4 x 4 grid. From a CT point of view the core idea is simple: click the square you want to slide into the empty space. The AgentCubes online solution is depicted in Figure 2. However, many computer program implementations, including visual programming languages [36], of the game exist which have upwards of 100-300 lines of code in languages like Python [37] and Java. If one were trying to learn Python and Java coding, implementing these solutions could have immense benefit. On the other hand, exposing students to CT necessitates tools that are better suited to solution expression without the addition of unnecessary complexity.



if empty →	then move →
if empty ←	then move ←
if empty ↑	then move ↑
if empty ↓	then move ↓

Fig. 2. 15 piece puzzle implementation in AgentCubes Online. Selecting a puzzle piece, e.g., the "1" will annotate the code to suggest the future state of the game.

C. *Supporting Execution & Evaluation*

The execution & evaluation stage can be supported by helping users to debug their programs as well change their misconceptions. Pea [21] describes debugging as:

> "…systematic efforts to eliminate discrepancies between the intended outcomes of a program and those brought through the current version of the program."

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 a Computational Thinking Tools to aid the debugging process. One strategy is to simply reduce the gap between solution expression and solution execution & evaluation. Punch cards are the classical negative example resulting in an extremely large gap. As this gap increases, users quickly loose sight of the causal relation between changes made to a program and manifestations of different behavior exhibited by running these changed programs [38]. Live programming [11] 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. Similarly, Conversational Programming [38] in AgentSheets [2] and AgentCubes online [12] extends the notion of live programming [11]. Even when a game is not running, by selecting an agent in the world, AgentCubes online will execute relevant code fragments one step into the future. It will annotate the code, specifying which rule will execute, in order to visualize potential discrepancies between the programs users have and the programs users want [21].

IV. Conclusions

While programming tools are aimed at programmers, Computational Thinking Tools have an educational mission and are aimed at users who want to become computational thinkers. To that end, Computational Thinking Tools must address not only syntactic but also semantic and pragmatic concerns of programming. Computational Thinking Tools need to support each of the three stages of the Computational Thinking process: problem formulation, solution expression, and solution execution/evaluation. By doing this, Computational Thinking Tools can support Computational Thinking in a variety of disciplines without resulting in unnecessary complexity.

V. Acknowledgements

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.

VI. References

[1] J. M. Wing, "Computational Thinking," Communications of the ACM, vol. 49, pp. 33-35, 2006.

[2] A. Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," presented at the 1996 IEEE Symposium of Visual Languages, Boulder, CO, 1996, 102-109.

[3] A. Repenning, D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, K. Gutierrez, and N. Repenning, "Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation," Transactions on Computing Education (TOCE), vol. 15, pp. 1-31, 2015.

[4] N. Escherle, S. Ramirez-Ramirez, A. Basawapatna, D. Assaf, A. Repenning, C. Maiello, Y. Endo, and J. Nolazco-Florez, "Piloting Computer Science Education Week in Mexico," presented at the Special Interest Group of Computer Science Education (SIGCSE 2016), Memphis, Tennessee, 2016.

[5] N. Escherle, D. Assaf, A. Basawapatna, C. Maiello, and A. Repenning, "Launching Swiss Computer Science Education Week," presented at the Proceedings of the 10th Workshop in Primary and Secondary Computing Education (WIPSCE), London, U.K., 2015, 11-16.

[6] S. Papert, "An Exploration in the Space of Mathematics Educations," International Journal of Computers for Mathematical Learning, vol. 1, pp. 95-123, 1996.

[7] S. Grover and R. Pea, "Computational Thinking in K–12: A Review of the State of the Field," Educational Researcher, vol. 42, pp. 38-43.

[8] J. M. Wing, "Computational Thinking Benefits Society," in 40th Anniversary Blog of Social Issues in Computing vol. 2014, J. DiMarco, Ed., ed. http://socialissues.cs.toronto.edu/2013/01/40th-anniversary/: University of Toronto, 2014.

[9] T. Bell, F. Rosamond, and N. Casey, "Computer science unplugged and related projects in math and computer science popularization," in The Multivariate Algorithmic Revolution and Beyond, L. B. Hans, D. Rod, V. F. Fedor, and M. niel, Eds., ed: Springer-Verlag, 2012, pp. 398-456.

[10] R. Arnheim, Visual Thinking. Berkley: University of California Press, 1969.

[11] S. McDirmid, "Usable Live Programming," presented at the SPLASH Onward!, Indianapolis, Indiana, 2013.

[12] A. Repenning, D. C. Webb, C. Brand, F. Gluck, R. Grover, S. Miller, H. Nickerson, and M. Song, "Beyond Minecraft: Facilitating Computational Thinking through Modeling and Programming in 3D," IEEE Computer Graphics and Applications, vol. 34, pp. 68-71, May-June 2014.

[13] B. J. Reiser, "Scaffolding Complex Learning: The Mechanisms of Structuring and Problematizing Student Work," Journal of the Learning Sciences, vol. 13, pp. 273–304, 2004.

[14] C. L. Willis and S. L. Miertschin, "Mind tools for enhancing thinking and learning skills," presented at the Proceedings of the 6th conference on Information technology education, Newark, NJ, USA, 2005, 249-254.

[15] B. A. Nardi and J. R. Miller, "The Spreadsheet Interface: A Basis for End User Programming," presented at the INTERACT 90 - 3rd IFIP International Conference on Human-Computer Interaction, Cambridge, UK, 1990, 977-983.

[16] A. A. diSessa, "An Overview of Boxer," Journal of Mathematical Behavior, pp. 3-15, 1991.

[17] K. Kahn, "Seeing Systolic Computations in a Video Game World," in Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, 1996, pp. 95-101.

[18] S. Papert, The Children's Machine. New York: Basic Books, 1993.

[19] S. Turkle, Evocative Objects: Things We Think With. Cambridge, USA: MIT Press, 2007.

[20] A. Repenning and A. Ioannidou, "AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D," presented at the IEEE Symposium on Visual Languages and Human-Centric Computing 2006, Brighton, United Kingdom, 2006, 27 - 34.

[21] R. Pea, " LOGO Programming and Problem Solving," presented at the 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., 1983.

[22] K. H. Koh, A. Basawapatna, V. Bennett, and A. Repenning, "Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning," presented at the Conference on Visual Languages and Human Centric Computing (VL/HCC 2010), Madrid, Spain, 2010, 59-66.

[23] A. Michotte, The Perception of Causality. London: Methuen & Co. Ltd., 1963.

[24] J. M. Wing, "Computational thinking and thinking about computing," Philosophical Transactions of the Royal Society, vol. 2008, pp. 3717-3725, 2008.

[25] K. H. Koh, A. Repenning, H. Nickerson, Y. Endo, and P. Motter, "Will it Stick? Exploring the Sustainability of Computational Thinking Education Through Game Design," presented at the ACM Special Interest Group on Computer Science Education Conference (SIGCSE 2013) Conference, Denver, Colorado, USA, 2013, 597-602.

[26] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," Communincation of the ACM, vol. 52, pp. 60-67, 2009.

[27] M. Conway, S. Audia, T. Burnette, D. Cosgrove, K. Christiansen, R. Deline, J. Durbin, R. Gossweiler, S. Koga, C. Long, B. Mallory, S. Miale, K. Monkaitis, J. Patten, J. Pierce, J. Shochet, D. Staack, B. Stearns, R. Stoakley, C. Sturgill, J. Viega, J. White, G. Williams, and R. Pausch, "Alice: Lessons Learned from Building a 3D System For Novices," presented at the CHI 2000 Conference on Human Factors in Computing Systems, The Hague, Netherlands, 2000, 486-493.

[28] E. P. Glinert, "Out of flatland: Towards 3-d visual programming," in IEEE 2nd Fall Joint Computer Conference, 1987, pp. 292-299.

[29] F. P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, pp. 10-19, 1987.

[30] "Women Who Choose Computer Science – What Really Matters, The Critical Role of Encouragement and Exposure," Google, Google Report, Technical reportMay 26, 2014, 2014.

[31] G. Fischer, "Domain-Oriented Design Environments," in Automated Software Engineering. vol. 1, ed Boston, MA: Kluwer Academic Publishers, 1994, pp. 177-203.

[32] B. Nardi, A Small Matter of Programming. Cambridge, MA: MIT Press, 1993.

[33] M. Guzdial, "Education: Paving the way for computational thinking," Communications of the ACM, vol. 51, pp. 25-27, 2008.

[34] C. Lewis, "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery:," Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, Technical Report CU-CS-372-87, August, 1987.

[35] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," Communications of the ACM, vol. 47, pp. 47-52, 2004.

[36] F. Hermans and E. Aivaloglou, "Do Code Smells Hamper Novice Programming?," Delft University of Technology, Software Engineering Research Group, Delft University of Technology Report TUD-SERG-2016-06, 2016.

[37] A. Sweigart, Invent Your Own Computer Games with Python, A beginner's guide to computer programming in Python., 2010.

[38] A. Repenning, "Conversational Programming: Exploring Interactive Program Analysis," presented at the 2013 ACM International Symposium on New ideas, New Paradigms, and Reflections on Programming & Software (SPLASH/Onward! 13), Indianapolis, Indiana, USA, 2013, 63-74.