https://doi.org/10.52326/ic-ecco.2022/KBS.06

Non-linguistic Thinking as an Effective Tool for Innovation in Education

Leon Brânzan [1], ORCID: 0000-0003-0049-6176
[1] Dept. of Informatics and System Engineering, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, str. Studenţilor, 9/7, b. 3, MD-2045, Chişinau, Republic of Moldova, leon.brinzan@iis.utm.md, https://fcim.utm.md/

Abstract—**In higher education subjects are traditionally taught in the form of lectures, where teachers are required by the curriculum to cover a certain amount of content in order to prepare their students for subsequent courses or examinations. It has been observed that people struggle when they are required to memorize a lot of new information, this phenomenon is explained by the working memory theory [1], and the negative effect is amplified when teaching subjects like programming, with the links between categories of information that has to be memorized not clearly identifiable by beginners. Lecture material for programming courses often mixes language-specific information (keywords, syntactical rules, ready solutions), mathematical basis for a given solution (type theory, algorithm theory), hardware-specific limitations (computer memory management) etc. This often has drastic consequences for students' success in later courses that rely on material from previous courses [2].**

This paper argues, that the process of learning is not simply about transferring knowledge from teacher to student. In fact, knowledge does not have to be "existing in an objective manner" for subsequent transmission, it can also be "built in a constructive manner by the learner" [3]. Within a traditional educational process software engineering students find themselves in situations where text is used to reason about other forms of text: typically code examples are shown first, then the code's structure and syntax are explained. While there is intrinsic value in reading code written by experts, reading explanations of that code is much less effective than trying to reason about the structure and function of a program, and various features of a programming language. This paper will attempt to showcase several teaching techniques that don't utilize textual explanations (either partially or completely), putting forward the argument that non-linguistic presentations can be more effective in teaching, under certain conditions. Several methods of achieving this effect will be described, with the main goal of appealing to the student's ability for computational thinking.

Keywords*—*non-linguistic learning; software engineering; cognitive load theory; computational thinking

I. Content Delivery

Software engineering is a multifaceted discipline. Mastering it requires memorizing a lot of factual information, knowledge of at least one programming language, and learning a particular set of skills that enable the learner to tackle complex engineering tasks. Programming can only be learned by solving problems specifically designed to develop these kinds of skills. This makes efficiently teaching software engineering difficult. The traditional way of teaching programming heavily relies on content delivery. Subsequent memorization of information delivered in this way requires development of multiple neural links in the corresponding neural networks, ergo – active interaction with a given piece of information and deliberate dwelling on the result of that interaction are needed [4]. Designing and writing software is one of the most complex problems students can be tasked with. Software itself is often on the leading edge of any given industry's advancement. It follows that software engineering courses should be on the bleeding edge of education. In actuality, the opposite is true. High-profile courses, like Harvard University's CS50 program, while claiming to be designed "with the aims of making the content of the course more widely available and contributing to public understanding of innovative learning"[1], do not go too far from the traditional lecture form in their attempts to innovate learning, only supplementing information delivered during lectures with various visualization techniques, and only sporadically. They do not illustrate the relations between different concepts presented to students from one lecture to the next, leaving it up to students to infer those connections.

Lecture is a process of deconstructing knowledge by the teacher, negotiating between teacher and learner, and subsequently reconstructing it by the learner. The same applies to learning via textbooks which inherits all of the

[1] CS50 Syllabus, Harvard College, https://cs50.harvard.edu/college/2021/fall/syllabus/

drawbacks of the lecture-form with the added detriment of having no ability to appeal to the author for clarification. Lectures are becoming less and less effective [5]. The ideas and skills required to master programming cannot be learned by listening to a lecture or by reading a book (though they are learned about, which has its own value). However, methods of doing it using software solutions which take into account how the human brain actualizes abstract concepts and processes code have existed for decades but are rarely used in education.

This paper will provide several concrete examples of how software enables students to learn the "spirit of engineering and problem solving" in order to support its main claim, that non-linguistic forms of learning can be extremely effective, especially for disciplines like programming.

II. Text and meta-text

While it has been noted, that "relationship between code and language may be ontogenetic as well as phylogenetic", and that "[it] is hard to imagine how code in its current form could have been invented in the absence of language" [6], learning programming as an activity is not based solely on learning a programming language. It is primarily about solving logical tasks and then applying a programming language to recording resulting solutions in text form for subsequent reuse. Programming languages are primarily a tool for formalizing generalized solutions.

The process of thinking itself does not directly correlate with speech, only overlapping with it in some areas [7]. More specifically, instrumental thinking – the ability to understand mechanical joints and devise mechanical solutions for problems that are mechanical in their nature – is linked to concepts and speech to a much lesser degree. Actions become subjectively comprehended before being manifested in speech. The primary function of instrumental thinking does not lie in transferring of knowledge but in applying accumulated knowledge to problem solving [8]. It is this kind of thinking that educators in engineering need to foster among their students.

When text or speech is used in a learning environment, three of its functions must be considered. First, the writer/speaker's intention is to communicate thoughts and ideas using language. Second, his intention is to be understood exactly. Third, "[...]beyond the linguistic code, communication entails a special structure of embedded intentions (the intention that others understand one's intentions) and is based on cooperative principles by which interlocutors work together toward understanding each other" [9]. This last function can never be guaranteed to apply when communicating through text or speech, because when the source of information encodes ideas into natural language the ideation process of the person(s) receiving that information is intruded upon. It is for these reasons that lectures and books for the most part fail to efficiently teach complex concepts, of which programming is a primary example. A question then arises: is there an alternative, more efficient way to teach the aforementioned "spirit of programming"; can students be taught to think like software engineers before learning a programming language and writing a single line of code?

III. Computational thinking

The term "instrumental thinking" borrowed from psychology, while it applies to programming, is not directly linked to it because it had been in use before programming as activity fully emerged. Computational thinking – "the thought process involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out" [10] – will be used in this paper to refer to the kind of thinking that should be developed in engineering students. Computational thinking as a concept does not describe a new kind of thinking process in a neuro-biological sense, it is a specialized term substituting "instrumental thinking" that implies understanding of objective processes specific to the problem at hand. However, it has been stated that "computational thinking is conceptualizing, not programming. It describes a way of thinking at multiple levels of abstraction, not only the ability to program" [11][emphasis added]. Thus, programming is a subset of computational thinking, since computational thinking involves solution expression, at the same time contrasting itself with programming. That has other implications as well. Firstly, that "programming" as an activity is separated into several distinct stages: formulating a problem, expressing a solution (in mathematical notation, programming language etc.), executing, evaluating – some or all of which go under the aegis of computational thinking, which "complements and combines mathematical and engineering thinking" [12]. Each of these stages requires different strategies, concepts, and forms of knowledge; this would mean that learning to do each of them would require different approaches as well. Secondly, expressing a solution – recording a set of steps using natural or formal languages – itself requires a specific form of thinking. The nature of this process is more easily understood since it involves mapping ready instructions to a specific language's grammatical and syntactic rules. The thought processes behind the remaining two activities are not as easily defined.

What's important to point out is, it is not hands-on (empirical) experience that is responsible for developing knowledge but the nature of the experienced activity, because the human brain develops new pathways in response to acquiring new information [13]. Therefore, the chosen learning strategy bears the most importance in regards to the effectiveness of learning processes.

IV. NON-LINGUISTIC METHODS OF TEACHING

While it remains to be empirically confirmed whether non-linguistic forms of learning are more efficient than traditional forms, there are several important benefits of using non-linguistic methods of teaching which could be leveraged for an overall more efficient learning process, regardless of the medium: language agnostic learning solutions, conformity with the multimedia principle of delivering information, cognitive load theory-aware methods of teaching, reflection-based learning, reactive learning environments, inference-inductive activities. This is by no means an exhaustive list, and these characteristics are not exclusive or inherent to non-linguistic learning methods but all of them can be tapped into using non-linguistic forms of learning with the help of proper tools to enhance the learning process.

Using non-linguistic methods of transferring information has two immediate consequences. First, students are not limited by the need to have prior knowledge of a specialized language or notation, or the need to dedicate time to getting acquainted with a notation/language. Second, it could also be beneficial to transfer information without the use of text, instead planting ideas into the respective areas of the brain directly (similarly to how code does it, appealing to the multiple-demand system [14]). Working memory is limited in how much information it can hold onto at any one time. That amount can be looked at as cognitive load, “the cognitive effort (or amount of information processing) required by a person to perform [a] task” [15]. It can be associated with a specific topic, the way information or tasks are presented to a learner, or the work put into creating a permanent store of knowledge (a *schema*). Non-linguistic forms of learning reduce the amount of extraneous information students must sift through while performing a learning task, which enables them to focus on the information relevant to learning, reducing cognitive load.

Cognitive theory of multimedia learning (*CTML*) assumes that “the working memory processes verbalized and visual pictorial information in two separate channels” [16]. To leverage that inherent characteristic of working memory it is advised to use multimedia instructions to “maximize the amount of available mental resources” [16]. It has been empirically shown that “people learn more deeply from a multimedia message when extraneous material is excluded rather than included” [1]. Tools that conform to CTML limit the use of text and guide the learner’s focus by other means (for example, communicating essential information and relations about information via spatial arrangement) [17].

Reflection is another powerful teaching tool. “When learners reflect, the otherwise implicit knowledge becomes digested through active interpretation, questioning, and exploration” [18]. It is worth pointing out that reflection is widely used in modern-day programming for incrementally improving software features (analysis) and diagnosing problems in software products (debugging). It is a crucial skill for a software developer. Yet very little time is dedicated to teaching core concepts of debugging, analysis, and profiling[2] to beginners. Purpose-built non-linguistic learning tools could and should incorporate reflection into the learning process, since “it is essential to increase learning outcomes and the learner’s awareness of their own learning” [18].

Reactivity is omnipresent in computer games because it is one of the primary tools designers and programmers use to guide users during the gaming process. Consider this simple example: playing a tabletop version of Solitaire for the first time. If the player does not have a good grasp of the game’s rules and makes a mistake (places a card in an invalid position) he will not be aware of his mistake unless someone else points it out. Now replace the tabletop version with a software version of the same game. The rules did not change, but now if the player makes the same mistake, the game can react to it by notifying the player of that. In fact, developers can have checks for all possible invalid game states in place to prevent players from making any kind of mistakes. This is a great teaching tool because it does not require prior experience with the game from players. The game can be successfully completed by trial and error, simultaneously teaching players its rules. This principle could be utilized in educational software to teach certain aspects without requiring learners to read large volumes of text.

Inference is one of the primary functions by which humans receive information, especially in cases where “sensory data are scanty or ambiguous, or incongruities occur in perceptual situation” [19]. Naturally, inference plays a major part in the process of non-linguistic learning. Human communication is characterized by its “intentionality and cooperative processes, not by language alone” [9]. A structure of such intentions embedded into communication “makes it possible to infer meanings beyond explicitly conveyed language” [9]. There is a plethora of research data supporting the idea of the effectiveness of visual “displays” in promoting learning. Cognitive processing has several forms (Mayer’s “select-organize-integrate” model) that can be leveraged “to

[2] The process of gauging the amount of resources software applications require to run, usually employed to detect deficiencies in resource management.

afford different kinds of inferences" by using visual displays [17].

In this context selection refers to focusing on specific information in an instruction. It can be promoted by driving attention to one part of a message and omitting non-critical information. Organization refers to inferring relations between pieces of information. It is especially important for memorization since associations between new data and prior knowledge "facilitates retrieval from long-term memory", this process is guided by integration.

To summarize, significantly reducing or completely eliminating the reliance on text should be the primary objective of computer science educators, as this paper argues, and the methods described above can be used for that with great effectiveness.

V. Existing research

There are several academic examples of note that show work being done on the subject of teaching programming to children using visual media. Experiments conducted with middle school children by Adele Goldberg et al. at Stanford showed promise initially [20]. Using early implementations of the Smalltalk programming language scientists attempted to ingrain basic programming concepts into children's minds. In 1973 Alan Kay, the original author of Smalltalk, joined Goldberg's team of researchers to develop new approaches to children's computer education using bleeding edge computer software technology. He proposed Smalltalk as the basis for further educational experiments. To Kay it was apparent that "the children could [...] draw pictures on the screen, but there seemed to be little happening beyond surface effects" [20]. At the same time he recognized that teaching concepts of object-oriented design to children would be a dead end, since it was still a fresh idea alien even to seasoned programmers. An alternative approach that utilized a visual language for communicating concepts had to be developed. Kay called it *literacy*, "the content of this new kind of authoring literacy should be the creation of interactive tools by the children" [20].

During the experiment each group of students consistently had a few children that excelled at their tasks and managed to produce working software prototypes (albeit very limited in scope and features): a painting application, object-oriented illustration system, music score capture system, circuit design system, to name a few [20]. After several groups of children had gone through the training, researchers made a discovery that each group's progress didn't generalize well at all. That is to say, only a small number of children would produce something significant at the end of the course, while 80% of the children would struggle, because the knowledge wouldn't come to them naturally. Another compounding effect was the children's background, "children were chosen from the Palo Alto schools (hardly an average background) and we tended to be much more excited about the successes than the difficulties" [20]. The overall success of any given child wouldn't "extend into the future as strongly" as Kay and Goldberg had hoped [20].

What had been happening was Kay using his existing knowledge to generate ideas that were far from intuitive for beginners, in actuality students were struggling to see the links between going from one set of instructions to the next in Kay's examples. Kay later concluded, "[it] isn't enough to just learn to read and write. There is also a literature that renders ideas. Language is used to read and write about them, but at some point the organization of ideas starts to dominate mere language abilities. And it helps greatly to have some powerful ideas under one's belt to better acquire more powerful ideas" [20]. This is where he agrees with Elliot Soloway [21] in that the success for most students depends not on any particular features of a programming language, but on how easy it is for a beginner to *be able to think in the same way that good programmers think*. Programming concepts should be learned gradually over a prolonged period of time in order to build up the structures that provide forward-thinking capabilities required to design software solutions. Kay calls this ability *fluency* – the process of building mental structures that hide "the interpretation of the representations" [20], similar to how people that know how to read don't perceive written text as symbols but rather as the direct meaning behind the text.

VI. Examples

This part will describe a software product that conforms to the characteristics and methods listed above, very effectively employing them to promote non-linguistic learning. It is worth noting that this example was not built as dedicated educational software, which is interesting in itself. The best work on promoting computational thinking using innovative approaches is being done outside of education[3], while the opposite would be expected.

Baba Is You[4] is a computer puzzle game in which puzzles are solved using linguistics. The rules of the game are not explained through textual descriptions, instead they are presented for each puzzle as objects that form short phrases that define the relations between objects on the screen, and are a part of the playing space (see fig. 1).

3 https://www.zachtronics.com/zachademics/
4 https://hempuli.com/baba/

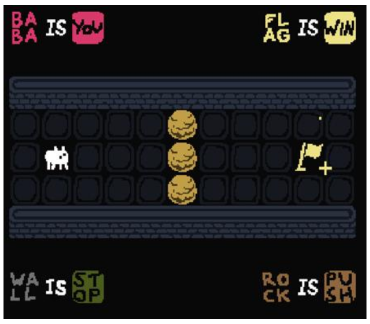

Figure 1. First puzzle in the series, all basic elements on screen

The text “BABA IS YOU” in the upper left corner indicates to the user that an entity named Baba is under his control, it is the player’s means of interacting with the world. “FLAG IS WIN” hints at the winning condition (get Baba to the flag to win), “WALL IS STOP” informs the player that the playable character cannot go through wall tiles. “ROCK IS PUSH” indicates to the player that tiles that look like rocks can be pushed away. But it is not until the second puzzle that the players realize, there is a lot more to the problems presented by the game to them (see fig. 2).

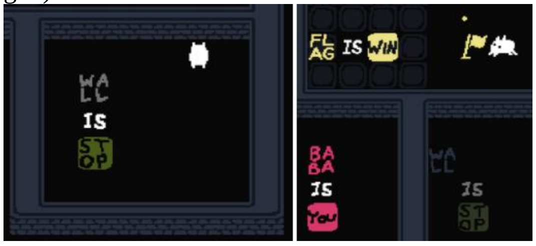

Figure 2. Starting conditions (left); winning condition satisfied (right)

Baba is surrounded by walls. The phrase “WALL IS STOP” is placed right next to the player’s character. This is a subtle hint at how subsequent puzzles are designed. Baba can interact with everything on the screen including phrases that describe the rules. And those phrases can be changed by using Baba to push parts of them away. Altering the phrase in such a way will change the rules governing the current puzzle. For example, pushing either “wall”, “is” or “stop” in that phrase will make all the walls on the screen non-corporeal allowing Baba to reach the flag on the other side of the wall. Since there is no limitation on where you can push different elements within the confines of the screen, it is possible to push “win” in order to form the phrase “WALL IS WIN” (see fig. 3).

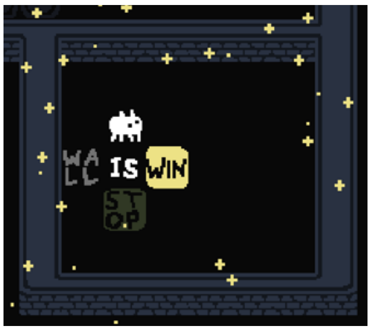

Figure 3. Changing rules to make “wall” the winning condition by spelling “wall is win”

This will expectedly allow the level to be completed by placing Baba on top of any wall tile, revealing an alternative solution to this puzzle. Players are taught another important skill in this instance: thinking “outside the box”, which is a particularly important skill for a programmer to have, because every problem in programming has multiple viable solutions. Some solutions are more effective, while other solutions – cheaper etc. But all of them are valid. The way *Baba is You* teaches this and other aspects of computational thinking without saying as much as an entire sentence, is ingenious.

Games are inherently suited for use in education. All higher animals engage in some form of games for learning purposes in the early stages of their lives [22]. Experience gained during such activities is applicable in real life. However, the same cannot be said about most computer gaming software. Games are well suited for learning in context of specific forms of knowledge, where games fare much better than other forms of media. Knowledge that is inherently hard to verbalize makes a good use case for educational software. Software that models systems is particularly good at teaching computational thinking, and excels at teaching programming. It enhances acquisition of skills like empirical validation, technical intuition etc.

VII. CONCLUSIONS

Before an engineering student can learn complex abstract concepts, he must learn a programming language and how to write a simple program in that language. The traditional approach to training using text is much less effective because the student is hindered by his lack of knowledge of: the relation between hardware and software, programming languages, basic constructs (algorithms), core paradigms etc. The use of specialized software allows for teaching those concepts to students without any prior knowledge, in parallel to other established methods.

Interactivity is one of the most important properties of computer software. It enables software to react to the actions of the user in ways that allow the user to gain experience and knowledge non-linguistically. Each interaction can be viewed as a self-contained experiment: the user thinks of a desirable outcome, tries an action, looks at the reaction, evaluates the outcome and repeats the loop if necessary. Coincidentally this mirrors thought processes that occur when completing programming tasks. Contemplating an idea, implementing it in code, launching it on a computer, the computer instantly reacting to it. If it reacts in an unexpected way it is seldom not a teaching moment, it enables *learning from expertise*. The user learns something new about the programming language being used, about the way the computer processes information, about their own thought process. These characteristic properties of computer software could be harnessed for educational purposes.

Software engineering is an applied science. It requires expertise in multiple domains. “Expertise is an ability acquired mostly by experience” [3]. However, it is worth noting that this effect is not uniform with learners across all levels of experience. “When assessing which agent, either the instructor or the learner, was most effective, we observed mixed results in the literature, [...] novice students may learn better under instructor-managed conditions, whereas more expert students may learn more under learner-managed conditions” [23]. Software has the means to provide education with the tools required for enabling a richer learning experience, circumventing traditional text-heavy forms of teaching, providing more effective methods of learning concepts that are essential for developing computational thinking, that at the same time are hard to verbalize. However, it is unclear to what extent non-linguistic new forms of teaching would be more effective. There is still need to accumulate empirical proof to quantify the assumed positive effects of non-linguistic learning, but that would have to be the subject of a future study.

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their feedback and advice.

REFERENCES

[1] R. Mayer, and L. Fiorella, “Principles for reducing extraneous processing in multimedia learning: Coherence, signaling, redundancy, spatial contiguity, and temporal contiguity principles”, Cambridge Handbook of Multimedia Learning, pp.279-315, 2014.
[2] M. Olsson, and P. Mozelius, “Learning to Program by Playing Learning Games”, European Conference on Games Based Learning, 2017.
[3] G. Albano, and F. Formato, “E-learning from Expertize: a Computational Approach to a non-textual Culture of Learning”, Advanced Learning Technologies Conference, 2001.
[4] “You Can Grow Your Intelligence”, Brainology Curriculum Guide for Teachers, Mindset Works, 2014.
[5] C. I. Petersen, P. Baepler, A. J. Beitz, and J. Walker, “The Tyranny of Content: “Content Coverage” as a Barrier to Evidence-Based Teaching Approaches and Ways to Overcome It”, CBE life sciences education, 2020.
[6] Y.-F. Liu, J. Kim, C. Wilson, and M. Bedny, “Computer code comprehension shares neural resources with formal logical inference in the fronto-parietal network”, eLife, 2020.
[7] L. Vygotsky, “Thinking and Speech”, State Socio-economic Publishing, Moscow, Leningrad, 1934, p. 88 (in Russian).
[8] Y. Kornilov, and I. Vladimirov, “Instrumental experience as component of the experience of practical change”, Yaroslav Psychology Herald, ed. 16, RPO, Moscow, Yaroslavl, 2005, pp. 21-28 (in Russian).
[9] U. Liszkowski, “Three Lines in the Emergence of Prelinguistic Communication and Social Cognition”, Journal of Cognitive Education and Psychology, vol. 10, no. 1, Springer Publishing Company, 2011, pp. 32-43.
[10] J. M. Wing, “Computational Thinking Benefits Society”, Social Issues in Computing, New York Academic Press, 2014.
[11] J. M. Wing, “Computational Thinking”, Communications of the ACM, vol. 49, no. 3, 2006, pp. 33-35.
[12] A. Lamprou, and A. Repenning, “Computational Thinking [does not equal] Programming”, Swissinformatics Magazine, 2017.
[13] E. R. Oby, M. D. Golub, J. A. Hennig, A. D. Degenhart, E. C. Tyler-Kabara, B. M. Yu, S. M. Chase, and A. P. Batista, “New neural activity patterns emerge with long-term learning”, PNAS, 2019.
[14] A. Ivanova, S. Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O’Reilly, M. U. Bers, and E. Fedorenko, “Comprehension of computer code relies primarily on domain-general executive brain regions", eLife, 2020.
[15] D. Shibli, and R. West, “Cognitive load theory and its application in the classroom”, Impact Journal of the Chartered College of Teaching, Making Learning Stick: Open Access Cognitive Science, 2018.
[16] M. Thees, S. Kapp, M. P. Strzys, P. Lukiwicz, J. Kuhn, and F. Beil, “Effects of augmented reality on learning and cognitive load in university physics laboratory courses”, Computers in Human Behavior, 2020.
[17] M. McCrudden, and D. N. Rapp, “How Visual Displays Affect Cognitive Processing”, Educational Psychology Review, 2017.
[18] J. Villareale, C. F. Biemer, M. S. El-Nasr, and J. Zhu, “Reflection in Game-Based Learning: A Survey of Programming Games”, preprint, 2020.
[19] M. D. Vernon, “Cognitive Inference in Perceptual Activity”, British Journal of Psychology, vol. 48, no. 1, 1957, pp. 35-47.
[20] A. C. Kay, "The Early History of Smalltalk", History of Programming Languages II, Association for Computing Machinery, 1996.
[21] E. Soloway, J. C. Spohrer, “Studying the Novice Programmer”, Lawrence Erlbaum Associates, Inc., New Jersey, 1989.
[22] J. Huizinga, “Homo Ludens”, Progress, 1992, pp. 21-45 (in Russian).
[23] J. C. Castro-Alonso, B. B. de Koning, L. Fiorella, and F. Paas, “Five Strategies for Optimizing Instructional Materials: Instructor- and Learner-Managed Cognitive Load”, Educational Psychology Review, 2021.