2017

A Conceptual Framework for a Software Development Process based on Computational Thinking

Catherine Higgins
TU Dublin, catherine.higgins@tudublin.ie

Ciaran O'Leary
Technological University Dublin, ciaran.oleary@tudublin.ie

Orla Hanratty
Technological University Dublin, orla.hanratty@tudublin.ie

See next page for additional authors

Recommended Citation

Higgins, C., O'Leary, C., Hanratty, O., Mtenzi, F. (2017). A conceptual framework for a software development process based on computational thinking. *Proceedings of the 11th International Technology, Education and Development Conference (INTED17), Valencia, Spain*, pp.455-464. ISBN: 978-84-617-8491-2 doi: 10.21125/inted.2017.024

A CONCEPTUAL FRAMEWORK FOR A SOFTWARE DEVELOPMENT PROCESS BASED ON COMPUTATIONAL THINKING

Catherine Higgins[1], Fredrick Mtenzi[1], Ciaran O'Leary[1], Orla Hanratty[1], Claire McAvinia[1]

[1]Dublin Institute of Technology (IRELAND)

Abstract

A software development process is a mechanism for problem solving to help software developers plan, design and structure the development of software to solve a problem. Without a process to guide the structured evolution of a solution, it is extremely likely that at least some aspect of the resulting software will be omitted or incorrectly implemented. Even though the importance of utilising a software process for solving problems is accepted in the business and academic communities, it is a topic that is addressed very lightly (if at all) in most freshman undergraduate computing courses with most courses focussing on programming procedures rather than the process of how to develop a solution. A consequence of this is that some students go on to develop maladaptive cognitive practices where they rush to implement solutions to problems with little planning. Typically these maladaptive practices involve surface practices such as coding by rote learning and cutting and pasting code from existing projects. Such practices can be very difficult to unlearn and can result in students lacking skills in planning and designing solutions to problems which can persist to graduation.

Despite these issues, little active research has been found on the development of software processes aimed at freshman third level learners and consequently there are few approaches available to help freshman students through all stages of the software process. However, there is a wealth of current research into computational thinking (CT) as a mechanism to help solve computational problems. Even though CT is seen as a key practice of computer science, most of the research into CT (as a named area) is aimed at 1st and 2nd level education with CT being a more implicit part of third level computing courses. This suggests that there is an exciting opportunity to explicitly exploit the affordances and skills of CT into a software process aimed at freshman third level learners.

This paper presents work which has been carried out as part of an ongoing research project into this issue in which the key skills associated with computational thinking are incorporated into a conceptual framework which will provide a structure for a software process aimed at freshman undergraduate computing students. This research is not tied to any particular programming paradigm but its use is assumed to be in the context of imperative, commercial programming languages. The framework is centred on declarative knowledge (in the form of threshold concepts) and procedural knowledge (in the form of CT skills) scaffolding freshman software development from initial planning through to final solution. The framework known as Computational Analysis and Design Engineered Thinking (CADET) – once operationalised as a software process with an accompanying support tool - aims to support the structured development of both software and student self-efficacy in the topic.

Keywords: computational thinking, programming, software process, introductory programming courses, threshold concepts.

1 INTRODUCTION

A software development process is a mechanism which informs a software developer of the steps and stages involved in developing quality software from initial analysis to final design and implementation [1]. There are many different categories of development process such as waterfall, spiral and agile which differ in how they approach software development but all share the same basic activities of analysis, design, coding, testing and debugging. Teaching these activities to students in a structured way is vital in enabling them to develop a proficiency in software development [2]. However, even though there are many software development processes available for experienced developers, very little work has been carried out on creating appropriate processes for freshman, third level learners [3]. This presents a vacuum for educators which means that software analysis and design is typically taught very informally and implicitly on introductory courses at third level [4]. Many courses have an emphasis on teaching a programming language and concepts as opposed to the teaching of how to plan and design

a solution before implementation [4-7]. Equally, this is reflected in undergraduate text books which have a focus on programming constructs rather than software development processes [8].

Unless they are guided, novices have a natural inclination to focus on the surface features of a problem in order to fit what is known about the problem into their existing knowledge [9]. Novices will often jump straight into implementing some aspect of a solution without any planning because they can find it difficult to separate ideas for solutions from the implementation of those ideas [10, 11]. This can lead to novices adopting maladaptive cognitive practices in software development, particularly surface practices (e.g. coding by rote learning) which can be very difficult to unlearn and can ultimately prohibit student progression in the acquisition of software development skills [12]. Therefore, it follows that if a software development process is incorporated explicitly in an appropriate way into introductory courses to scaffold students in software development practices, this could limit the development of maladaptive practices.

This paper describes a conceptual framework which has been devised as a route to providing such a software development process using the affordances of computational thinking. Section 2 gives a rationale for the framework by describing related research into computational thinking, problem solving and threshold concepts, each of which contributes to the framework. Section 3 describes the framework while section 4 concludes the paper with a discussion of the contribution this research aims to make to research in software engineering education.

2 BACKGROUND AND RELATED RESEARCH

Despite the wealth of established software development processes aimed at experienced developers working on large projects, there is very little available for introductory third level computing courses. Most research into development processes for such courses have a specific focus on a particular aspect of the process. Examples are the STREAM process [3] which focuses on design in an object oriented environment; the P^3F framework [13] with a focus on software design and which aims to arm novice designers with expert strategies; a programming process by Hu et al [14] which focuses on generating goals and plans and converting those into a coded solution via a visual block-based programming language; POPT [15] which has a focus on supporting software testing and Morgado & Barbosa's process [2] which aims to support students through all developmental stages with the use of template forms coupled with an instructor supplied prototype to aid understanding of the problem. The framework described in this paper is similar to Morgado & Barbosa's process in that it aims to support all aspects of developing software but the focus here is based on the provision and utilisation of developmental constructs and problem solving skills to guide the development of software solutions. It is envisaged that any software development process based on this framework will be adaptable to students' experience so the process can continue to be used by students even after they leave the novice stage of learning.

Learning how to develop software requires both declarative knowledge (programming concepts) and procedural knowledge (skills needed to apply the declarative knowledge to create a working solution) [16, 17]. Equally, it is well known that affective issues have a strong impact on student learning particularly in the domain of software development [18]. Therefore, any framework used as a basis to support a software development process for novice third level learners must incorporate the following components:-

1. An identification of key declarative knowledge which must be taught if students are to become competent software developers (irrespective of programming language).
2. An identification of key procedural knowledge used to apply the declarative knowledge when solving problems. This involves the skills associated with computational problem solving.
3. An identification of the key affective issues which can affect student learning.

The following sub-sections describe each of these components in detail.

2.1 Threshold Concepts to provide Declarative Knowledge

In order to identify the key concepts that should be included in the framework, it is important to restate that this framework should be applicable to a variety of imperative, commercial languages taught in introductory courses. Therefore, the content in the framework should not be language specific but should

capture the essential content that forms the basis of such languages in a generic fashion. The theory of threshold concepts offers a route to identifying and codifying such essential content.

Threshold concepts (TCs) have been proposed as a way of characterising the progression of learning by students in any subject area. Meyer and Land [19, 20] suggest that within each subject, there are certain concepts that are difficult to grasp and are "troublesome" but once mastered, they act as a portal that can open up a new way of thinking within the subject area that is transformative and irreversible. They also state that "crossing the threshold" to understanding these concepts is a necessary step if students are to advance and become proficient in the subject area. In examining the research carried out into TCs for computing, it is clear that there isn't a consensus on a list of TCs for this area. Extensive research into identifying TCs in computing have been mainly carried out by a group of European researchers [21-25] with other researchers also attempting to identify TCs in computing [26-31]. When examining all the available research on the identification of TCs for computing, there were two concepts that emerged as having consensus. The first is *program dynamics* (phrase coined by Sorva [29]) meaning having a mental understanding of the dynamics of state and program/algorithm execution. The second concept– only relevant in an object oriented paradigm - is *object behaviour*. It can also be argued that object behaviour is a subset of the concept of program dynamics (when the development paradigm is object oriented) as understanding object behaviour requires having a mental model of the behaviour and state of "live" objects in execution.

Therefore, the declarative knowledge for this framework will focus on concepts that allow students to learn and understand about *program dynamics* and secondly (and optionally), an understanding of *object behaviour* when the development paradigm is object orientation.

2.2 Problem Solving and Computational Thinking

There are several suggested methodologies in the literature when it comes to solving problems. Many of these are very generic and can be applied to any kind of problem [32, 33]. However, in the context of this research project, one of the more appropriate strategies for general problem solving that can be adapted to a wide variety of educational contexts - including software development - is found in Pólya's problem solving strategy [34]. This text postulated that there are four main stages to problem solving (where the context was mathematical problem solving) which are 1) Understand the problem; 2) Make a plan to solve the problem; 3) Carry out the plan, and 4) Reflect on the success of the plan.

In the context of computational problem solving, these stages have been adapted in this research as 1) Understand the problem, 2) Break problem into tasks, 3) Design and code each task, 4) Evaluate task solution and learning.

In order to ustilise this problem solving strategy, students must be taught appropriate problem solving skills that will allow them to carry out each stage. Computational Thinking (CT) is an approach to problem solving that aims to utilise the strategies undertaken by expert software developers as a mechanism to solve problems in any discipline that require a computational solution [35, 36]. CT was popularised by Jeanette Wing [37] in her seminal paper although the phrase was originated by Seymour Papert [38] in the context of using the affordances of computer mediated instruction to help students experiment with - and understand - mathematical concepts. A clear distinction is made in the literature between CT and programming in that even though programming is a key route to enable CT, the process of CT itself may be applied to solving problems that do not involve a programming solution [39].

The research interest in CT has grown as a result of our ubiquitous technological environment. In such an environment, it is argued that CT is a basic literacy which should be an integral part of the educational process across all educational levels and disciplines where students of all ages can become adept at seeing how problems can be broken down and solved with appropriate computing tools in innovative ways [40]. As a result, there is a lot of active and very interesting research into the teaching and assessment of CT. Much of this research is aimed at first and second level education [40] often in the context of block/visual languages (such as Scratch, Alice, AgentSheets) [39, 41] and game based learning which is used to teach CT skills to all levels including third level [5, 42]. At undergraduate level, CT is also being researched as a mechanism to introduce computer science skills to non-computing students [43]. While there is no doubt that the key skills associated with CT are an inherent part of computer science courses [44] which principally teach commercial, imperative languages [45, 46]; there is little evidence in the literature to show that the advantages of CT are being explicitly exploited and incorporated into mainstream introductory software development courses at third level. This suggests

that there is an opportunity for computing courses to explicitly mine the affordances of CT to support introductory education at third level.

Given that research into CT has only been active for the past decade, the topic is still relatively immature and so there is yet little agreement on what constitutes a detailed definition of CT or how it can be assessed [40, 47, 48]. In that context, it should be noted that the focus of this paper is not to enter the debate regarding a definition for CT but rather to utilise the commonly associated skills associated with the term. From the literature, the principal skills most commonly cited are: applying abstraction; problem decomposition; data analysis and representation; pattern recognition; writing algorithms, modelling and simulation; testing and debugging [47, 49-51]. Other concepts such as iteration, recursion and parallel thinking are also suggested by Wing [37]. It is the contention of this researcher that iterative and recursive thinking are concepts which form part of declarative knowledge which is then made procedural by the CT skills of writing algorithms, modelling and simulation. It is also felt that parallelism is a concept which is too advanced for novice computing students studying in the context of imperative, textual languages.

The framework described in this paper will utilise the skills commonly associated with CT to provide the procedural knowledge required by students learning how to solve software problems using Pólya's adapted problem solving model.

2.3 Affective Learning

Affective learning focuses on developing students' belief systems, emotions and attitudes. One affective factor important in determining the success of students in a software development course is the students' beliefs in their self-efficacy in the subject [52]. Self-efficacy is a judgment by a person of how capable they are of successfully carrying out a task which influences their choice of activities, the amount of effort they expend, their persistence in overcoming difficulties and ultimately their performance outcomes [52, 53]. Individuals with strong self-efficacy believe that they can succeed in challenging tasks which means that they are motivated to attempt such tasks and have confidence as they begin the journey to completing the task. Studies in the theory of self-efficacy has found it to be malleable meaning that it can be changed particularly in introductory courses to improve performance [54]. Suggested routes to improving self-efficacy include supplying clear instruction of the strategies required at each step of the problem-solving process, presenting mastery models to demonstrate how to carry out a task competently and confidently, supplying coping models to make explicit the difficulties students can experience in carrying out software development and giving useful and positive feedback [55]. While a strong belief system can push learners to exceed their expectations, it's also important that they don't have a distorted view of their actual capabilities [53]. Therefore, its important that students are able to reflect accurately on their learning and experiences to allow them understand what they know, don't know and ultimately gain accurate self-efficacy in their abilities. Students should be helped develop their metacognitive skills to enable them to monitor, evaluate and regulate their own learning strategies which is an important ability for the development of independent learning, encouraging students to take more responsibility for their learning [56]. Therefore, giving students awareness of their self-efficacy through metacognition and providing an environment to promote a growth in self-efficacy is a guiding force in the future deployment of this framework.

3 COMPUTATIONAL ANALYSIS AND DESIGN ENGINEERED THINKING FRAMEWORK

The term "*computational design thinking*" is currently used in architecture and construction as that industry has moved from computer-aided-design (CAD) to more sophisticated computational processes. It refers to the fact that the affordances of advanced computer technology is creating new types of design processes and new ways of thinking about design [57].

In the context of this research, the term is extended to "*computational analysis and design engineered thinking*" (CADET) to refer to the art of software development processes being computer mediated so that the stages of the process produce concrete visual artefacts with which students can experiment, evaluate and reflect on as part of their learning. These artefacts should be minimal and should adopt a "just enough" approach to allow students arrive at solutions in a supported fashion. Therefore, the

CADET approach is about learning computational concepts and skills in a computational thinking focused problem solving environment where the final work product is a verified program.

The CADET framework – summerised in figure 1 – borrows the categories used in the CT framework as suggested by Brennan and Resnick [50] which are *Concepts*, *Practices* and *Perspectives.* These categories are used as they fit the intersection of threshold concepts and computational thinking. The context for Brennan and Resnick's framework is the Scratch programming environment aimed at the K-12 profile.

Concepts (threshold stages)	Practices (CT skills)	Perspectives (affective issues)
TC1. State and Sequential Flow TC2. Non-Sequential Flow Control TC3. Modularity TC4. Object Behaviour	CT1. Abstraction CT2. Data Representation CT3. Decomposition CT4. Evaluation of solutions (Testing, Debugging, Critiquing) CT5. Pattern recognition CT6. Generating Algorithms (modelling and simulation)	AI1. Self-Efficacy

Figure 1 - CADET Framework (Source: Author)

In the context of this paper, the **concepts** contain the declarative knowledge which are categorised as the threshold concepts of *program dynamics* and *object behaviour* identified in section 2.1 which drive the instructional content for novice learners. In order to teach the two threshold concepts in a manageable way to students, they have been broken down into four threshold concept stages which have been codified in column 1 of figure 1 as TC1 - TC4. TC1 – TC3 represent *program dynamics* with TC4 relating to *object behaviour*. This fourth stage needs only be included if operating in an object oriented paradigm. A description of each of these threshold concept stages is as follows:-

TC1 : State and Sequential Flow – This concept involves gaining an understanding of "simple" data items (e.g. characters, numbers, strings) and how their state changes when sequential actions are carried out on them.

TC2: Non-sequential Flow Control – This concept keeps the focus on state but will add complexity to this idea by presenting more complex actions such as iterative and conditional actions and how they affect state and flow control.

TC3: Modularity – This concept involves the use of modularity and how that affects state and especially flow control.

TC4: Object Behaviour – This concept examines the idea of objects and their state. It also looks at the connection between state and behaviour and sees how objects interact and activate each other's behaviour.

The **practices** used in this context are the computational thinking skills introduced in section 2.2 of this paper and are codified as skills CT1 – CT6 in column 2 of figure 1. When operationalised into a software development process, the computational thinking skills will be acquired though repeated problem solving based on the adapted problem solving model described in section 2.2.

CT1: Abstraction – This is the ability to view some aspect of a problem solution at various levels of detail. In computer science, abstraction is often used as a mechanism to simplify a problem by allowing the developer to zoom in or out of a construct to reveal as much detail as is relevant. For example, an information system being developed for a college with several degree programmes may award prizes of various magnitude to students with the highest average examination grade in each programme. Abstraction could be employed in this case to allow the developer to focus on generating a partial solution which orders the prize winning students from best to worst performance in order to assign appropriate prizes to them. The detail about each prize winning student's internal grades or how they were chosen as the best student in their programme is kept hidden to allow the developer focus on the task they are currently working on without getting overwhelmed with excessive and unnecessary detail. In this framework, abstraction is considered to a skill that can be employed at every stage of the software

development process from analysis of the problem through to the design of a solution and coding that solution. Therefore, any software development process that results from this framework will provide a mechanism to allow abstraction to be utilised to maximum effect at every stage of the process.

CT2: Data Representation – Central to developing software is the ability to identify the core data that needs to be stored as part of the resulting program. Being able to identify this data and know how it should be represented is a key skill that is acquired through practice.

CT3: Decomposition – When solving a problem, a core skill is the ability to decompose the problem into smaller and more manageable sub-tasks that need to be carried out. In this framework, it is envisaged that acquiring the skill of decomposition also requires the skill of integration as each sub-task that is designed and implemented will then need to be integrated into a final working software solution.

CT4: Evaluation of Solutions – The ability to evaluate solutions is crucial in software development. This evaluation comes in the form of:-

- Testing analysis, design and coding artefacts for correctness and consistency across artefacts.
- Debugging any errors that are found.
- Critiquing solutions on the basis of software quality characteristics such as usability, non-duplication of effort and effective utilisation of appropriate threshold concepts.

CT5: Pattern Recognition – This refers to the ability to recognise that all or part of an exact or similar problem has been solved before. Recognising patterns is an extremely valuable skill that minimises duplication of effort and enables reuse of analysis, design and coding artefacts.

CT6: Generating Algorithms – An algorithm is a high-level representation or model of a proposed design for a problem that gives clear instructions on how the problem will be solved. There are many formats that an algorithm can take – they can be represented as diagrams (e.g. flow-charts), prototypes, as pseudo-code or in another programming language. The ability to generate an algorithm (or use existing algorithms) is a crucial skill in software development as it gives developers a mechanism to design a solution at a high-level without having to worry about the nuances of the particular programming language that will be used to implement the final solution. It also provides a mechanism to discuss potential solutions with other interested parties who may not be programmers. For students learning how to develop software it is a particularly important skill to acquire as without it they will have no option but to try and implement a solution through trial and error.

All 6 of the skills in this framework are largely similar to those suggested by Brennan and Resnick but while they are grouped together by those researchers, here they are not grouped as it is important to see them as separate skills. For example, it is possible to utilise abstraction without utilising modularisation. Other specific features of the practices presented in this CADET framework are:-

- The ability to determine the data that is required to solve a problem is considered to be an essential skill (and not a concept);
- Pattern recognition represents reuse but not just at the code level which implies other artefacts created as part of the analysis and design stages of the software development process should be reusable also;
- The skills of generating algorithms and problem decomposition are included as essential skills.

Finally, **perspectives** are the affective issues associated with learning software development which are classified in section 2.4 as being embodied in self-efficacy. This is an issue that will need to be measured in any resulting software development process via student refection.

4 DISCUSSION

This paper presents a framework which is the first stage of an ongoing research project into providing a software development process aimed at freshman undergraduate computing students. Despite the recognised importance of software development processes, this research has identified a gap in software engineering education in the provision of appropriate software development processes for this cohort of students. This paper aims to contribute to this field by presenting a novel framework which

combines current research into Computational Thinking as a problem solving process underpinned by the focus of Threshold Concepts in order to support students who are learning how to develop software solutions from problem specification through to the final tested product. The aim of the framework is to provide scaffolding to students in the form of a structured development process to improve competency and self-efficacy in software development. It is the contention of this research that the provision of such a process - while not a silver bullet to eradicate all of the problems students experience in learning software development - would provide a structured and scaffolded environment to directly address the maladaptive cognitive habits that students often form and find hard to unlearn.

The next stage of this project involves designing a software development process based on this framework for novice third level learners. An accompanying support tool will also be developed to support the software development process by providing students with a platform where they will carry out extensive problem solving via the creation of traceable, visual artefacts from problem specification through to the final software product. This tool will form the basis on which the process can be evaluated via an action research methodology. A description of the software development process and the results from its evaluation will be published in future papers.

REFERENCES

[1] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*, pp. 12-29: ACM, 2006.

[2] C. Morgado and F. Barbosa, "A structured approach to problem solving in CS1," presented at the Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, Haifa, Israel, 2012.

[3] M. E. Caspersen and M. Kolling, "STREAM: A First Programming Process," *Trans. Comput. Educ.,* vol. 9, no. 1, pp. 1-29, 2009.

[4] J. W. Coffey, "Relationship between design and programming skills in an advanced computer programming class," *J. Comput. Sci. Coll.,* vol. 30, no. 5, pp. 39-45, 2015.

[5] C. Kazimoglu, M. Kiernan, L. Bacon, and L. MacKinnon, "Developing a game model for computational thinking and learning traditional programming through game-play," J. Sanchezand K. Zhang, Eds., ed. Chesapeake, USA: AACE, pp. 1378-1386, 2010 .

[6] C.-C. Liu, Y.-B. Cheng, and C.-W. Huang, "The effect of simulation games on the learning of computational problem solving," *Computers & Education,* vol. 57, no. 3, pp. 1907-1918, 2011.

[7] S. Xiaoyuan, "Toward more effective strategies in teaching programming for novice students," *Teaching, Assessment and Learning for Engineering (TALE), 2012 IEEE International Conference on,* pp. T2A-1-T2A-3, 2012.

[8] A. K. Whitfield, S. Blakeway, G. E. Herterich, and C. Beaumont, "Programming, disciplines and methods adopted at Liverpool Hope University," *Innovation in Teaching and Learning in Information and Computer Sciences,* vol. 6, no. 4, pp. 145-168, 2007.

[9] L. J. Ball, T. C. Ormerod, and N. J. Morley, "Spontaneous analogising in engineering design: a comparative analysis of experts and novices," *Design Studies,* vol. 25, no. 5, pp. 495-508, 2004.

[10] V. Kokotovich, "Problem analysis and thinking tools: an empirical study of non-hierarchical mind mapping," *Design Studies,* vol. 29, no. 1, pp. 49-69, 2008.

[11] R. J. Fornaro, M. R. Heil, and A. L. Tharp, "What Clients Want - What Students Do: Reflections on Ten Years of Sponsored Senior Design Projects," *19th Conference on Software Engineering Education & Training (CSEET'06),* pp. 226-236, 2006.

[12] T.-C. Huang, Y. Shu, C.-C. Chen, and M.-Y. Chen, "The development of an innovative programming teaching framework for modifying students' maladaptive learning pattern," *International Journal of Information and Education Technology,* vol. 3, no. 6, p. 591, 2013.

[13] D. R. Wright, "Inoculating Novice Software Designers with Expert Design Strategies," in *American Society for Engineering Education*, 2012: American Society for Engineering Education, 2012.

[14] M. Hu, M. Winikoff, and S. Cranefield, "A process for novice programming using goals and plans," presented at the Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136, Adelaide, Australia, 2013.

[15] V. L. Neto, R. Coelho, L. Leite, D. S. Guerrero, and A. P. Mendon, "POPT: a problem-oriented programming and testing approach for novice students," presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[16] A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education,* vol. 13, no. 2, pp. 137-172, 2003.

[17] L. Thomas *et al.*, "A Broader Threshold: including skills as well as concepts in computing education," in *Fourth Biennial Conference on Threshold Concepts: from personal practice to to communities of practice*, pp. 154 - 158: NAIRTL, 2014.

[18] S. Hansen and E. Eddy, "Engagement and frustration in programming projects," presented at the Proceedings of the 38th SIGCSE technical symposium on Computer science education, Covington, Kentucky, USA, 2007.

[19] R. Land, G. Cousin, J. H. F. Meyer, and P. Davies, "Threshold concepts and troublesome knowledge (3): implications for course design and evaluation," *Improving student learning diversity and inclusivity,* pp. 53-64, 2005.

[20] J. H. F. Meyer and R. Land, "Threshold concepts and troublesome knowledge: linkages to ways of thinking and practicing," Edinburgh, University of Edinburgh, 2003.

[21] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander, "Putting threshold concepts into context in computer science education," in *ACM SIGCSE Bulletin*, vol. 38, pp. 103-107: ACM, 2006.

[22] J. Boustedt *et al.*, "Threshold concepts in computer science: do they exist and are they useful?," in *ACM SIGCSE Bulletin*, vol. 39, pp. 504-508: ACM, 2007.

[23] K. Sanders *et al.*, "Student understanding of object-oriented programming as expressed in concept maps," *SIGCSE Bull.,* vol. 40, no. 1, pp. 332-336, 2008.

[24] K. Sanders *et al.*, "Threshold concepts and threshold skills in computing," presented at the Proceedings of the ninth annual international conference on International computing education research, Auckland, New Zealand, 2012.

[25] R. McCartney, A. Eckerdal, J. E. Mostrom, K. Sanders, and C. Zander, "Successful students' strategies for getting unstuck," presented at the Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, Dundee, Scotland, 2007.

[26] E. Vagianou, "Program working storage: a beginner's model," presented at the Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006, Uppsala, Sweden, 2006.

[27] J. T. Khalife, "Threshold for the introduction of programming: providing learners with a simple computer model," *28th International Conference on Information Technology Interfaces, 2006.,* pp. 71-76, 2006.

[28] D. Shinners-Kennedy, "The Everydayness of Threshold Concepts: State as an Example from Computer Science," in *Threshold Concepts within the Disciplines*Rotterdam: Sense Publishers, pp. 119-128, 2008.

[29] J. Sorva, "Reflections on threshold concepts in computer programming and beyond," presented at the Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli, Finland, 2010.

[30] P. Alston, D. Walsh, and G. Westhead, "Uncovering “Threshold Concepts” in Web Development: An Instructor Perspective," *Trans. Comput. Educ.,* vol. 15, no. 1, pp. 1-18, 2015.

[31] K. Sanders and R. McCartney, "Threshold concepts in computing: past, present, and future," presented at the Proceedings of the 16th Koli Calling International Conference on Computing Education Research, Koli, Finland, 2016.

[32] J. D. Bransford and B. S. Stein, "The IDEAL problem solver," 1993.

[33] G. Alexander and B. Renshaw, "Supercoaching," *London: Business Books,* 2005.

[34] G. Polya, *How To Solve It*, 2nd ed. Princeton University Press., 1957.

[35] J. M. Wing, "Computational thinking and thinking about computing," *Philosophical transactions of the royal society of London A: mathematical, physical and engineering sciences,* vol. 366, no. 1881, pp. 3717-3725, 2008.

[36] L. Mannila *et al.*, "Computational Thinking in K-9 Education," presented at the Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference, Uppsala, Sweden, 2014.

[37] J. M. Wing, "Computational thinking," *Communications of the ACM,* vol. 49, no. 3, pp. 33-35, 2006.

[38] S. Papert, "An exploration in the space of mathematics educations," *International Journal of Computers for Mathematical Learning,* vol. 1, no. 1, pp. 95-123, 1996.

[39] S. Y. Lye and J. H. L. Koh, "Review on teaching and learning of computational thinking through programming: What is next for K-12?," *Computers in Human Behavior,* vol. 41, pp. 51-61, 2014.

[40] F. Kalelioglu, Y. Gülbahar, and V. Kukul, "A Framework for Computational Thinking Based on a Systematic Research Review," *Baltic Journal of Modern Computing,* vol. 4, no. 3, p. 583, 2016.

[41] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR),* vol. 37, no. 2, pp. 83-137, 2005.

[42] K. S. Marshall, "Was that CT? Assessing Computational Thinking Patterns through Video-Based Prompts," *Online Submission,* 2011.

[43] L. Perković, A. Settle, S. Hwang, and J. Jones, "A framework for computational thinking across the curriculum," in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pp. 123-127: ACM, 2010.

[44] P. J. Denning, "The profession of IT: Beyond computational thinking," *Commun. ACM,* vol. 52, no. 6, pp. 28-30, 2009.

[45] R. M. Siegfried, D. Greco, N. Miceli, and J. Siegfried, "Whatever happened to Richard Reid's list of First Programming Languages?," *Journal of Information Systems Education,* vol. 10, no. 4, p. 7, 2012.

[46] S. Xinogalos, "Object-Oriented Design and Programming: An Investigation of Novices' Conceptions on Objects and Classes," *ACM Transactions on Computing Education (TOCE),* vol. 15, no. 3, p. 13, 2015.

[47] D. Barr, J. Harrison, and L. Conery, "Computational thinking: A digital age skill for everyone," *Learning & Leading with Technology,* vol. 38, no. 6, pp. 20-23, 2011.

[48] S. Grover, R. Pea, and S. Cooper, "Designing for deeper learning in a blended computer science course for middle school students," *Computer Science Education,* Article vol. 25, no. 2, pp. 199-237, 2015.

[49] ISTE and CSTA, "Computational Thinking in K–12 Education leadership toolkit," *http://csta. acm. org/Curriculum/sub/CurrFiles/471.11 CTLeadershiptToolkit-SP-vF. pdf>. Acesso em,* vol. 10, p. 12, 2014.

[50] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, pp. 1-25, 2012.

[51] L. A. Gouws, K. Bradshaw, and P. Wentworth, "Computational thinking in educational activities: an evaluation of the educational game light-bot," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pp. 10-15: ACM, 2013.

[52] S. Wiedenbeck, "Factors affecting the success of non-majors in learning to program," presented at the Proceedings of the first international workshop on Computing education research, Seattle, WA, USA, 2005.

[53] A. Bandura, "Self-efficacy," in *Encyclopedia of human behavior.* vol. 4, V. S. Ramachaudran, Ed., ed. New York: Academic Press., pp. 71-81, 1994.

[54] M. E. Gist and T. R. Mitchell, "Self-Efficacy: A Theoretical Analysis of Its Determinants and Malleability," *The Academy of Management Review,* vol. 17, no. 2, pp. 183-211, 1992.

[55] A. Sewell and A. St George, "Developing efficacy beliefs in the classroom," *The Journal of Educational Enquiry,* vol. 1, no. 2, 2009.

[56] J. Sheard, A. Carbone, D. D'Souza, and M. Hamilton, "Assessment of programming: pedagogical foundations of exams," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pp. 141-146: ACM, 2013.

[57] A. Menges and S. Ahlquist, *Computational Design Thinking: Computation Design Thinking*. John Wiley & Sons, p 10, 2011.