A MODERN PEDAGOGICAL CONTENT KNOWLEDGE APPROACH TO THE TEACHING OF COMPUTER PROGRAMMING IN SCHOOLS

By

ENGR. AKPORUME DAVIDSON
Department of Computer Science,
College of Education,
Warri.

Abstract

The purpose of this study is to uncover the Pedagogical Content Knowledge (PCK) of Computer Science Education, with focus on Programming. PCK has been defined as the knowledge that allows teachers to transform their knowledge of the subject into something accessible for their students. The core questions to uncover this knowledge are: what are the reasons for teaching programming?; what are the concepts taught in programming?; what are the most common difficulties/misconceptions students encounter while learning to program; and how can this topic be taught?. Some of the answers found are, respectively: enhancing students' problem solving skills; programming knowledge and programming strategies; general problems of orientation; and possible ideal chains for learning computer programming.

The 21st century is characterized by the ubiquitous presence of technology in everyday life. New generation students are surrounded by computer related instruments and will possibly do a job that has not been invented yet. Computing succeeded to conquer most of the aspects of our society and in order to fit in, people need to be versatile and adaptable to modern and future technology. This scenario emphasised the need to provide an education that can offer students and future adult the ability to understand and work with computer related instruments. Holmbe, Melver and George (2001) evidenced the existing need to broaden the effort of computer science educators to contribute to the knowledge of why computer science should be taught, what topics at all, how computer science should be taught, and for whom the teaching of computer science education is meant. The focus of this paper is on providing answers to these questions relative to a specific topic of computer science education, that is, programming.

The answer to the four questions introduced lead to the understanding of the concept called Pedagogical Content Knowledge (PCK), (Shulman, 1986). PCK is a concept that combines the knowledge of the content (e.g., mathematics, computer science, etc.) to the knowledge of the pedagogy (e.g., how to teach mathematics, how to teach computer science, etc.), giving insights into educational matters relative to the learning and teaching of a topic. Teachers with good PCK are teachers who can transform their knowledge of the subject into something accessible for the learners.

Methodology Adopted for this Study

The instrument of this study is the conceptual framework introduced by Grossman (1989, 1990) as it schematizes the PCK through simple and easy to use questions such as: why teach a certain subject?, what should be taught?, what are learning difficulties?, and how to teach?.

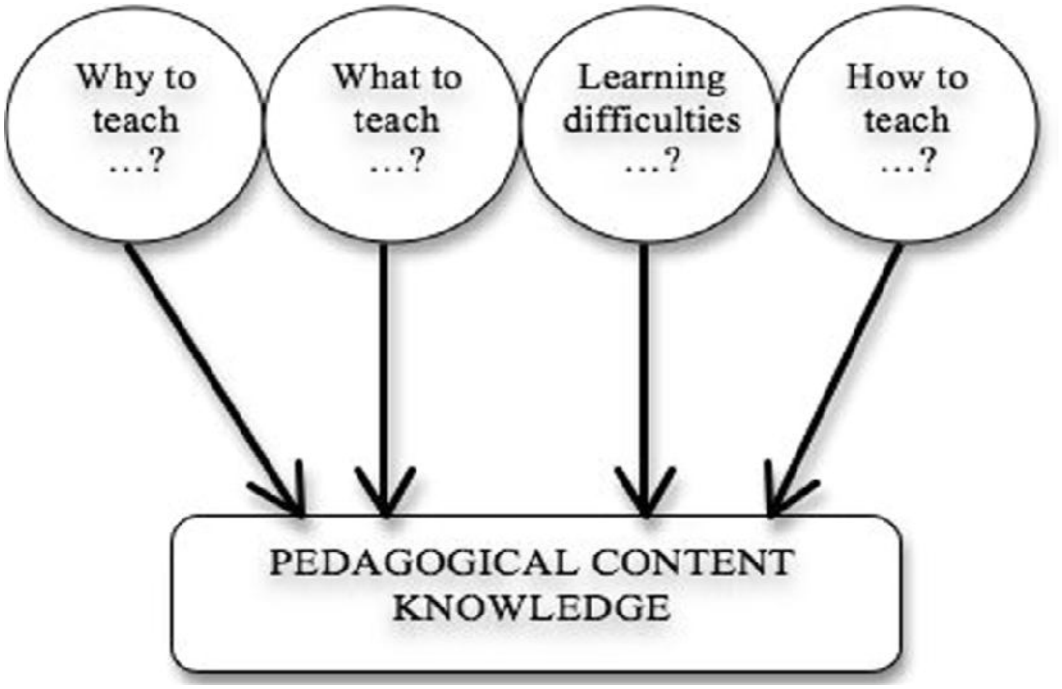

Figure 1: Diagram of Grossman's Conceptual Framework of PCK

Pedagogical Content Knowledge (PCK)

According to Shulman (1986), PCK includes "the most useful forms of representation of topics, the most powerful analogies, illustrations, examples, explanations, and demonstrations - in a word, the ways of representing and formulating the subject that make it comprehensible to others. Pedagogical content knowledge also includes an understanding of what makes the learning of specific topics easy or difficult: the conceptions and preconceptions that students of different ages and backgrounds bring with them to the learning of those most frequently taught topics and lessons."

Saeli, Perrenet, Jochems and Zwaneveld (2011) stated that there is in fact, a difference between knowing how to program and being able to teach programming. The classroom, where learning and teaching occur, is a complex environment in which several processes and actions happen. But when talking about PCK, a special attention should be spent on students' learning. An aspect of PCK concerns the need teachers have to represent and formulate the subject, so that comprehension can occur. Research findings confirm that different learners have different learning styles (Rayner and Riding, 1997), and needs. This implies that there are no single most powerful forms of representation, the teacher must have at hand a veritable armamentarium of alternative forms of representation, some of which derive from research whereas others originate in the wisdom of practice.

According to Saeli, et al (2011), an example in computer science could be the teachers' knowledge about the concept of programming structures, and the need to formulate their knowledge in a way that can be easily understood by their students. All research in this domain agrees on claiming that PCK is a knowledge that develops with years of teaching experience (Rovegno, 1992; Grossman and Lynn, 1990).

PCK of Programming

The PCK of a subject is the knowledge that enables researchers and teachers to better understand the issues related to the teaching and learning of the subject and consequently, provide a better teaching. In this section, preliminary answers to the core questions uncovering the PCK of programming are provided, using the method described above.

Why Teach Programming to Secondary School Students?

Soloway (1993) answered this question by arguing that in learning to program one learns powerful problem-solving/design/thinking strategies. This is because when students program, they first need to find a solution to a problem, and then they need to reflect on how to communicate their solution to the machine, using syntax and grammar, through an exact way of thinking (Papert, 1980; Szlávi and Zsakó, 2006). The latter contributes to the students' natural language skills because they are required to learn to tell, in an unambiguous way, what they want the computer – an unintelligent machine – to perform
(Hromokovic, 2006).

Programming involves the ability to generate a solution to a problem. Generating solutions means that one of the learning outcomes is the ability to solve problems and also, if the problem is a big problem, the ability to split the problem into sub problems and create a generalizable central solution. In addition, the student achieves the ability to create usable, readable and attractive solutions.

Problem solving skills can be deployed to solve "realistic" problems in various domains together with the computing goals (Sims-Knight and Upchurch, 1993; Dagiene, 2005). Transferability of these and other skills is the argument that brought Feurzeig and his colleagues (Feurzeig, Papert, Bloom, Grant, Solomon, 1970) to introduce programming as a way to help students to understand mathematics concepts such as: rigorous thinking, variable, function, decomposition, debugging and generalization.

Syslo and Kwiatkowska (2006) went further by exploring those mathematics concepts that can benefit from programming, but which have not be included in the (Polish) secondary school curriculum yet. Besides transferability of skills, when learning to program, students also acquire a sense of mastery over a technological instrument and establish contact with some of the deepest ideas of different disciplines such as: science, mathematics and the art of intellectual model building (Papert, 1980). Moreover, programming is a new generation subject, which brings together pieces from different areas such as: linguistics, mathematics and economics (Mulder, 2002). This completeness gives students the opportunity to be faced with a multi-disciplinary subject that connects different aspects in a single class. Students could experience the opportunity to delve deeper into previously acquired knowledge.

What Should be Taught?

By answering this question we aim to understand what are the core concepts of programming students need to learn. Decisions about what it is, that needs to be taught are usually taken by curriculum and examinations designers. In Computer science, efforts to define a suitable curriculum have been made since the late '60s (Atchison, Conte, Hamblen, Hull, Keenan, Kehl, 1968). However, the different curricular representations (Van den Akker and Voogt, 1994) should be considered, including: the ideal curriculum, which refers to the original ideas and intentions of the designers; the formal curriculum, denoting the written curriculum (documents, materials); the perceived curriculum, indicating the interpretation of the users, especially the teachers of the curriculum; the operational curriculum, identifying the actual instruction process in the classroom; and the experiential curriculum, which represents students' reactions and outcomes.

According to Romeike (2008), the core of programming is all about problem solving and creating a program as solution. In programming, two kinds of knowledge can be distinguished, namely program generation and program comprehension (Van Merriënboer and Krammer, 1987; Robins and Rountree, 2003; Mannila, 2007). In the first case, the programmer analyzes the problem, produces an algorithmic solution, and then translates this algorithm into a program code. This means that students should be coached in the process of problem solving, reflection on this process, and in the development of algorithmic ways of thinking (Feurzeig, Papert, Bloom, Grant and Solomon, 1970; Resnick and Ocko, 1990; Sims-Knight and Upchurch, 1993). As for program comprehension, the programmer is asked to give a demonstration of his/her understanding of how a given program works. The teaching in secondary school of program generation and program comprehension is considered very important. Programs are a set of instructions that computers execute in order to perform a task and are written in a programming language.

In the process of learning to program, Govender (2006) identified from a technical point of view, three main aspects students need to learn: data, instructions and syntax. Data refers to the concepts of variables and data types for procedural programming and objects involving attributes and actions for object oriented (OO) programming. As for instructions, the needed understanding is about control structures and subroutines for the procedural programming, and interacting objects and methods in the case of OO programming. Syntax denotes the group of rules that determine what is allowed and what is not within a programming language. Syntax rules determine what is called the vocabulary of the language, how programs can be constructed using techniques such as loops, branches and subroutines.

Govender's classification, however, does not take care of the modularity and abstraction aspects of programming, as for example Abelson and Sussman (1996) do. They identify three main aspects: primitive expressions, representing the simplest entities that a language is concerned with; means of combination, by which compound elements are built from simpler ones; and means of abstraction, by which compound elements can be named and manipulated as units. These three aspects deal with two kinds of elements: data and instructions. By using these three mechanisms in combination with each other, it is possible to formulate complex programs, starting from simpler ones.

A final aspect, equally important, is the semantic of a program, also referred to as the meaning of a program. A semantically correct program is a program that performs the required task. Programs written with different syntax can perform the same semantic task.

What are the Learning Difficulties of Beginning Programmers?

Because of the complexity of individuals, different students will have different needs and difficulties. It is well known that many students have difficulties in learning programming. Programming is a very complex subject that requires effort and a special approach in the way it is learned and taught (Van Diepen, 2005; Govender, 2006) and often, novice programmers hold misunderstanding and misconceptions. In early stages of the learning process, a correct program often results as an unexpected surprise (DuBoulay, 1989).

DuBoulay (1989) identified five kinds of difficulties/areas which have a certain degree of overlap in the learning/teaching of programming, and these are:

(i) **Orientation**, finding out what programming is useful for and what the benefits of learning programming are.

(ii) **The Notional Machine** understanding the general properties of the machine that one is learning to control and realizing how the behaviour of the physical machine relates to the notional machine.

(iii) **Notation**: which includes the problems of aspects of the various formal languages such as syntax and semantics.

(iv) **Structures**, understanding the schemas or plans that can be used to reach small-scale goals (e.g., using a loop).

(v) Mastering the pragmatics of programming (learning the skill to specify, develop, test and debug a program using the available tools).

From a student-computer relationship point of view, Pea (1986) identified the existence of persistent conceptual language-independent “bugs” in how novices program and understand programs. The starting point of the analysis of conceptual “bugs” is that students have a tendency to *converse* with a computer as if it were a *human* (considered also as the superbug), with consequences such as expecting the computer to interpret students’ conversations. Pea (1986), distinguished three different kinds of conceptual “bugs”: the *parallelism* bug, which refers to the assumption that different lines in a program can be active or somehow known by the computer at the same time, or in parallel. Another bug is the *intentionality bug*, for which students believe that computers “go beyond the information given” in the lines of programming code being executed when the program is run. The last bug, *egocentrism*, refers to students’ assumption that there is more of their meaning for what they want to achieve in the program than is actually present in the code they have written (e.g., “Don’t print what I say, print what I mean!”). Students’ conceptions do not guide their attention to consider these problems as relevant reasons for their programs not working as planned.

Another problem students could face is the paradigm shift (Kölling, 1999; Mazaitis, 1993) in cases where their teacher proposes them to learn more than one programming language with different paradigms (e.g., procedural and object oriented), although this is not advisable in an introductory course at secondary school level. Students usually encounter problems in passing from one paradigm to another, especially from the procedural to the object oriented (but not the vice versa).

Regarding the acquisition of problem solving skills, Ginat (2006) and Weigend (2006) agreed that students tend to maintain local, limited-insight points of view of the problem, leading often to undesirable and erroneous outcomes. Also, Weigend (2006) observed, even when finding a mental or practical solution to a problem, students fail to write a correct program that does the job. The reason might be that students are not trained to translate mental intuitions in a communicative way, or might be connected with the semantic of a program.

Semantic is also considered to be a problematic aspect of programming. This is because it requires the student to put together different parts of a program (variables, expressions, statements, control structure, objects and methods) into a working solution. Semantic is closely related to the debugging activity and the related correctness of a program (Pea and Kurland, 1983), a concept introduced by Dijkstra (1968, 1972). When teachers choose a programming language offering a more complex syntax, students will be faced with both semantic and syntax difficulties (Mannila, Peltomaki & Salakoski, 2006).

How should Programming be Taught?

In attempting to provide answer to this question, efforts are made to understand what the best approaches to introduce students into the learning of programming are, not only to prevent the above mentioned difficulties/misconceptions, but also to hook students' motivation in an effective and engaging way.

Hromovic (2006) suggested that programming be seen as a skill to communicate in an unambiguous way, a set of instructions to an unintelligent computer. If this process could take place by means of a relatively simple programming language offering a simpler syntax than other commonly used programming languages, students could focus more on the semantic aspect of the program and produce fewer syntax errors (Mannila *et. al.*, 2006).

Another way to start this learning process could be by the use of practical examples, such as rewriting recipes for cooking for a cooking machine (Hromovic, 2006). The process should lead students to write at first, simple programs, and then combine the simple solutions together to obtain solution to more complicated problems (Abelson and Sussman, 1996). This approach has the twofold purpose to let the student not only experience the historical development, but also learn the concept of modularity and reusability. Writing a set of instructions to solve a problem is the definition of algorithm. To achieve algorithmic thinking students should solve many problems which should be chosen independently from any programming language (Futschek, 2006), and should follow some pedagogical principles (Romeike, 2008). In fact, algorithmic thinking can be successfully introduced without the aid of a computer at all (Curzon and McOwan, 2008). However, it happens that students fail to translate their correct reasoning into an unambiguous set of instructions for the machine. To overcome this, students could be coached in analysing their intuitions and connecting them to the designated task (Weigend, 2006).

Linn and Dalbey (1989) suggested an ideal chain for learning computer programming, which gradually goes from program comprehension and ends with program generation. The chain has three main links: single language features, design skills, and general problem-solving skills. According to Linn and Dalbey (1989), the ideal chain should start with the understanding of the language features, knowledge that can be assessed by asking students to reformulate or change a language feature in a program so that the program does something slightly different. The second link of the chain consists of design skills, which are a group of techniques used to combine language features to form a program. This chain link also includes templates (stereotypical patterns of code that use more than a single feature) and procedural skills (used to combine templates and language features in order to solve new problems, including planning, testing and reformulating). The third link of the chain, problem-solving skills, is useful for learning new formal systems. Problem-solving skills can be assessed by asking students to solve problems using an unfamiliar formal system such as a new programming language. Though this chain of cognitive accomplishment requires an extensive amount of time, it forms a good summary of what could be meant by deep learning in introductory programming (Robins, Rountree & Rountree, 2003).

To provide students with a framework for understanding, some model or description of the machine should be introduced, where a model should be designed around each group of amateur learners, distinguished either for their age, background or kind of studies (Du Boulay, O'Shea & Monk, 1989). Students working with such models excelled at solving some kind of problem more than students without the model (Mayer, 1989). An example could be the metaphor of a black box inside the glass box as a way to present computing concepts to novices. The reason is that novices start programming with very little idea of the properties of the notional machine implied by the language they are learning.

The previous approaches mostly deal with the difficulties and misconceptions presented in the previous section. A look at approaches which aim at teaching programming in an engaging way, reference should be made to the family of programming environments and suited programming languages developed with the main goal of introducing students to the programming practice in active and motivating scenarios. These environments have been specially designed to ameliorate the difficulties students usually encounter when learning to program with normal programming languages (Mannila et al., 2006). The list is quite long and the first efforts have been already made in the early '70s. Among the most popular are Logo and its derivates (Feurzeig et al., 1970; Papert, 1980), initially designed to teach mathematics, which has the focus to enhance problem solving skills; Scratch (Curzon & McOwan, 2008) which, based on a metaphor of building bricks and offering much of the same functionality as Logo, allows students to create syntactically correct program, and leaves the students to focus on the semantic aspect; and finally the more modern Alice Greenfoot and Gamemaker (Cooper, Dann, & Pausch, 2003; Kölling and Henriksen, 2005). These learning environments find their basis in Piaget's model of children's learning where students are fostered to build their own intellectual structures, if provided with the right material. It is then the teacher's task to find suitable support/stimuli/learning material to use with each of these tools. Some of these languages and environments, however, might not include some structures or topics important to the learning of programming (Murnane and McDougall, 2006).

Conclusion

There are different reasons why learning programming is inherently difficult. Programming requires not a single, but a set of skills. Those skills form a hierarchy and a programmer will be using many of them at any point in time. The most important skill for beginning programming students is to develop their problem solving abilities. This means developing a computational environment mainly based on problem solving activities from different domains. When the student reaches a higher competence level in generic problem solving, the environment starts to propose typical programming problems. This seems to be the best approach as programming education should be preceded by the development of a sound problem solving competence. The environment also tries to adapt itself to each particular student characteristics, namely taking into consideration her/his previous work and preferred learning style when selecting activities and interaction modes that will be used with that particular student.

The task in portraying the PCK of programming will be to find the answers not only from a general point of view (programming in general), but from the perspective of each of the most frequently taught topics, which are at the heart of learning to program (e.g., variables, functions, etc.). An example will be answering the four questions regarding the teaching of problem solving skills. Despite the fact that some of these answers are available for some concepts, most have still to be studied. Therefore, this paper calls for more research to portray the PCK of the most commonly taught programming topics.

References

Abelson, H., Sussman, G.J. (1996). *Structure and Interpretation of Computer Programs,* 2nd edition. Series

Almstrum, V.L., Guzdial, M., Hazzan, O., Petre, M. (2005). Challenges to computer science education research.

An, S., Kulm, G., Wu, Z. (2004). The pedagogical content knowledge of middle school, mathematics teachers in China and the U.S. *Journal of Mathematics Teacher Education*, 7, 145–172.

Cochran, K.F., DeRuiter, J.A., King, R. A. (1993). Pedagogical content knowing: an integrative model for teacher preparation. *Journal of Teacher Education*, 44, 263–272.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *SIGCSE 2003.*

Curzon, P., & McOwan, P. (2008). Engaging with computer science through magic shows. Paper presented at The 13th Annual Conference on Innovation and Technology in Computer Science Education.

Dagien˙e, V. (2005). *Teaching information technology in general education: challenges and perspectives.* In: R.T. Mittermeir (Ed.), Proceedings of International Conference on Informatics in Secondary Schools – Evolution and Perspectives, Klagenfurt, Austria, 53–64.

Dijkstra, E.W. (1968). A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8, 174–186.

Du Boulay, B. (1989). Some difficulties of learning to program. In: Soloway, E., Spohrer, J.C. (Eds.), *Studying the Novice Programmer*, London, Lawrence Erlbaum Associates, 283–299.

Du Boulay, B., O'Shea, T. Monk, J. (1989). The black box inside the glass box : presenting computing concepts to novices. In: Spohrer, C. (Ed.), *Studying the Novice Programmer*. London, Lawrence Erlbaum Associates, 431–446.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., Solomon, C. (1970). Programming-language as a conceptual framework for teaching mathematics. *Newsletter SIGCUE Outlook*, 4(2), 13–17.

Futschek, G. (2006). Algorithmic thinking: the key for understanding computer science. In: Mittermeir, R.T. (Ed.), ISSEP 2006, *LNCS*, 4226, 159–168.

Govender, I. (2006). Learning to Program, Learning to Teach Programming: Pre- and In-service Teachers' Experiences of an Object-oriented Language. University of South Africa.

Grossman, P.L., & Lynn, P. (1990). *The making of a Teacher: Teacher Knowledge and Teacher Education.* New York, Teachers College Press, Columbia University.

Hromkoviˇc, J. (2006). Contributing to general education by teaching informatics. In: Mittermeir, R.T. (Ed.), ISSEP 2006, *LNCS*, 4226, 25–37.

Linn, M.C., Dalbey, J. (1989). Cognitive consequences of programming instruction. In: Soloway, E., Spohrer, J.C. (Eds.), *Studying the Novice Programmer*. London, Lawrence Erlbaum Associates, 58–62.

Mannila, L., Peltomaki, M., Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, 16, 3, 211–227.

Mayer, R.E. (1989). The psychology of how novices learn computer programming. In: Soloway, E., Spohrer, J.C. (Eds.), Studying the Novice Programmer. London, Lawrence Erlbaum Associates, 129–159.

Mulder, F. (2002). van BÈTA – naar DELTA-discipline (in English: Computer Science: from a BÈTA to a DELTA subject). *Informatica, Tinfon*, 11, 48.

Papert, S. (1980). *Mindstorms. Children, Computers and Powerful Ideas*. Basic Books, Inc. Publishers, New York.

Rayner, S., Riding, R. (1997). Towards a categorisation of cognitive styles and learning styles. *Educational Psychology,* 17, 5–27.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education*, 13, 137–172.

Rovegno, I.C. (1992). Learning to teach in a field-based methods course: the development of pedagogical content knowledge. *Teaching and Teacher Education,* 8, 69–82.

Shulman, L.S. (1986). Those who understand: knowledge growth in teaching. *Educational Researcher,* 15, 4–14.

Shulman, L.S. (1987). Knowledge and teaching: foundations of the new reform. *Harvard Educational Review,* 57, 1–22.

Sims-Knight, J.E., Upchurch, R.L. (1993). *Teaching software design: A new approach to high school computer science.* Paper presented at The Annual Meeting of the American Educational Research Association. Atlanta, GA.

Soloway, E. (1993). Should we teach students to program? *ACM Communications*, 36, 21–24.

Soloway, E., Spohrer, J.C. (1989). *Studying the Novice Programmer*. London: Lawrence Erlbaum Associates.

Syslo, M.M. & Kwiatkowska, A.B. (2006). Contribution of informatics education to mathematics education in schools. In: Mittermeir, R.T. (Ed.), ISSEP 2006, *LNCS*, 4226, 209–219.

Turner-Bisset, R. (1999). The knowledge bases of the expert teacher. *British Educational Research Journal*, 25, 39–55.

Van den Akker, J. & Voogt, J. (1994). The use of innovation and practice profiles in the evaluation of curriculum implementation. *Studies in Education Evaluation*, 20, 503–512.