Program Working Storage: A Beginner's Model

Evgenia Vagianou

Computer Information Systems Department
The American College of Greece
6 Gravias str., Ag. Paraskevi
15342, Athens, Greece
jes@acgmail.gr

Department of Informatics
University of Sussex
Falmer, Brighton
BN1 9QH, UK
ev38@sussex.ac.uk

ABSTRACT

The aim of this paper is to introduce and validate the concept of *program working storage* (PWS) as a) a means of smooth transition of students in introductory programming courses from the *end-user stance* to the *programmer stance*, and b) a system which can provide comprehensive understanding of certain difficult programming concepts. In this respect, the program-memory interaction is considered as a possible "threshold concept" [31, 33]. Based on constructivism [16, 23, 41, 42], the PWS is then discussed as a potential beginner's viable model, which can be, later on, *refined* to what Ben-Ari describes as a viable computer model [5]. The extent to which the PWS can be used as a conceptual framework, which will enable teachers and learners to focus on program-memory interaction across a variety of dimensions, and eventually relate them to form a coherent whole, is also examined. The exact implementation of the PWS in the context of the various programming languages is beyond the scope of this paper. Nevertheless, it constitutes a topic for detailed study and future research.

Keywords

Teaching programming, introductory programming, constructivist instruction, preconceptions, threshold concepts.

1. INTRODUCTION

Computing curricula, in undergraduate institutions, are usually shaped with respect to the computing disciplines offered (i.e. CE, CS, IS, IT, SE), and the course targeting approach[1]. Due to cost-related issues, broad student audience strategies, and the fact that the strongest commonality across all disciplines is "concepts and skills of computer programming", a *programming-first* model is usually adopted [1,2]. Inevitably, introductory programming courses often serve as the actual starting point of study.

Empirical studies, relative to novice programmers and introductory programming courses, yield that students find the learning of programming particularly difficult [15, 27, 44]. Their prior knowledge and understanding of the domain (or related domains) affect the manner in which they engage learning [33]. An interesting, though expected, observation is the common perspective of students in introductory programming courses and end-users, who regard a program as the "magical instrument" that "will do the job". Frequently expressed with comments like "it didn't work" or "it did it again", the above perspective also implies the notion of a "someone" or "something" else being responsible for what is happening.

It seems safe to claim that in the mind of the end-user, "the program" (or "the computer") constitutes a distinct but irrelevant to his/her concerns ontology, since it only serves as a tool, while the point of interest is the task to be accomplished. Such conceptions can be justified as viable in the sense that they are consistent with the context in which they were created [40, 42]. Nevertheless, when novice programmers carry them in a new context, they can prove inefficient [36, 40] and even haphazard [4]. According to the constructivist theory of learning, though, preconceptions are crucial in that they form the basis for new knowledge to be built.

Educators of introductory programming courses are called to deprecate the *end-user stance* of their students and enforce the *programmer stance*, which must further comply with a more consistent *computing expert stance*[2]. This has to be achieved in a specified time period, which compared to the potentially vast amount of new information, concepts and perspectives that a student needs to perceive and interpret, is so inadequate that selection of focus is inevitable. The practical constrains of teaching objectives and overlapping course material are just the final "straws" on the pile.

The aim of this study is to introduce and validate the concept of *program working storage* (PWS) as a) a means of smooth transition of students in introductory programming courses from the *end-user stance* to the *programmer stance*, and b) a system which can provide comprehensive understanding of certain difficult programming concepts. In this respect, the program-memory interaction is examined as a possible "threshold concept" [31, 33], and the PWS is discussed as a potential beginner's viable model, which can be, later on, *refined* to a viable computer model [5]. The suggested approach, although constructivistic in nature, has a behaviouristic dimension, since one of its goals is that the PWS is eventually automatically recalled.

The PWS is also discussed from the perspective of a conceptual framework, which will enable teachers and learners to focus on program-memory interaction across a variety of dimensions (such as data types, implicit/explicit reference, states, etc.) and eventually relate them to form a coherent whole.

The suggested representation of a PWS is graphical with limited constrains to its precise deployed form.

[1] According to the CC2005 Computing Curricula Task Force Report, there are two possible approaches: the filter, which recommends parallel discipline-specific introductory course sequences, and the funnel, which recommends a common introductory sequence [1].

[2] Terms are discussed in section 2.4.

The exact implementation of the PWS in the context of the various programming languages is not in the scope of this paper and, therefore, not extensively addressed. Nevertheless, it constitutes a topic for detailed study and future research.

2. LITERATURE REVIEW

The literature related to this study encompasses research in a variety of fields, which, due to space limitations, are divided into three basic categories: novice programming issues, teaching and learning theories, mental models and graphical external representations.

2.1 Novice Programming Issues

The research in novice programmers' problems, misunderstandings, bugs etc., is very large and detailed [e.g. 9, 14, 24, 35, 36], and the literature addressing ways to cope with them is analogous [4, 14, 18, 20, 44]. This major concern of educators may be fairly attributed to two (2) reasons:

a) Programming involves specifying behaviour that will occur in the future [6, 7]. It, therefore, imposes the utilization of certain cognitive skills, which in the case of novice programmers are inert if present at all. In their study, Fix et al. [17] identify several characteristics in experts' mental models, which are not observable in novices' representations, thus implying the absence of the related skills.

b) Existing knowledge and experience lead to preconceptions which, as White [43] notes, easily turn to misconceptions. The main problem seems to be what Pea [36] describes as "conversing with a human", leading to language-independent "bugs". The issue is that natural language pragmatics, intuitively used in human interaction, contradicts with the "mechanistic" rules that a formal system interprets instructions.

There exist a really large number of approaches and tools, which have been developed in order to address the above points. In their majority they utilize a technique, known as "program visualization" [30, 39], which involves exposure of memory contents in order to facilitate program comprehension.

2.2 Teaching and Learning Theories

2.2.1 Behaviourism and Cognitivism

Traditional teaching approaches are based on behaviouristic and cognitive theories, the fundamental assumptions of which are in accordance with the objectivistic philosophical paradigm [16, 23]. Objectivism assumes the world and the mind as two distinct ontologies. The world is viewed as a complete, well-structured reality, and the mind as an abstract processing machine [16]. The goal of teaching is to efficiently "map the structure of the world onto the learner's mind" [16, 23], so that it "mirrors reality" [23].

2.2.2 Constructivism

Constructivism is a philosophical paradigm [16, 23, 41] concerned with the nature of knowledge. Without denying the existence of an objective world, knowledge is not assumed to be a part of it. The main argument is that knowledge is constructed by an individual's own experiences, thus forming one's personal *version* (model) of the world ("known world" [41]).

Constructivism, as a learning theory, encourages the building of knowledge on a subject, by changing the nature of the questions a student asks about it [41]. Usually regarding a subject as a complex system consisting of a number of sub-domains (contextual entities), educators have been deeply concerned with the conceptions (i.e. the models) that students carry in the various contexts. A model developed in a certain context, may prove inadequate in a new situation, resulting to unjustifiable inferences.

A considerable amount of research is concerned with ways of dealing with preconceptions. In their analysis of knowledge in transition [40], Smith et al. argue that the existing approaches[3] involve a significant shift from preconceptions to expert models, by dispelling the former and adopting the latter. They note the anti-pedagogical dimension of such practices[4], as well as, the conflict with the basic premise of constructivism – that knowledge is built "recursively". Eventually, they propose "refinement" as an effective approach. In this case, students acknowledge that their existing knowledge is inadequate to explain phenomena, and transform it into more sophisticated forms through relatively stable intermediate states of understanding.

In CS education, much of the research has been performed by Ben-Ari [3, 4]. One of his strongest points, regarding specifically CS education, is that a viable computer model must be present before programming is engaged. This suggestion is supported by prior research [14, 28, 29] and has been defended by recent research [18, 38, 44].

2.2.3 Threshold concepts

There is not much literature, yet, on "threshold concepts" (suggested by Meyer and Land [32, 33]) since it is a quite recent theoretical notion, the roots of which lie in constructivism. Threshold is characterized a difficult core concept of the domain to be studied, which, once understood, provides a broad ground for comprehension of more advanced concepts. Its originators shape the nature of such concepts based on five dominant characteristics [33]:

A threshold concept is "transformative", in the sense that it can provide a significantly improved understanding of the subject-matter.

It is "irreversible", in that, once the perspective it will provide is understood, it is "unlikely to be forgotten".

It is "integrative", in that it exposes the hidden interrelatedness of something.

It may be "bounded", in the sense that the conventional semantics of the language may lead to substantially different inferences.

It may be "troublesome", in that it may involve troublesome knowledge (i.e. "ritual", "inert", "conceptually difficult", "alien", or "tacit" [33]).

The research for CS threshold concepts has only started.

2.3 Mental and External Representations

2.3.1 Mental Models

In the cognitive science literature, there are two (2) main perspectives concerning the nature of mental representations. The theory of "Formal Rules" [11, 12] assumes that humans

[3] "Replacement", "confrontation", "overcoming" [40].

[4] I.e. assessing one's perception as fundamentally wrong.

construct propositional representations, independently of the form of the stimuli that may emerge them. Therefore, the most efficient representational form is considered to be the sentential. In any other case, the objects of the domain will need to be converted to nouns before encoded, which implies extra mental effort.

The alternative theory, "Mental Models" [21, 22], argues that the entities of mental models represent both structure and content and, therefore, may have arbitrary or iconic properties. It is further argued that the structure of a mental model, which is considered its most fundamental property, should be identical to the structure of the spatial relations as they are perceived or conceived.

Based on the above, Boudreau and Pigeau [10] performed an empirical study, in order to test the effectiveness of spatial reasoning using diagrammatic and sentential representations. Their outcomes point that in both cases, humans perform more or less the same spatial reasoning task. Nevertheless, diagrams were significantly favoured in terms of easiness of use and efficiency.

2.3.2 Graphical External Representations

Graphically expressed external representations (ERs) address primarily concept visualization, i.e. "the process of forming a mental image or vision of something not actually present to the sight" [37]. Cognitive scientists have extensively studied the value of developing ERs and their effect on learners' understanding, while their pay-offs, mostly obvious in the special case of diagram use, have been associated with three main information-processing operations [26]: a) correspondence between elements is automatically expressed, b) information needed for the same inference, can be grouped, thus, reducing the amount of search required to locate the information, and c) *perceptual inferences* are supported, which, by utilizing the power of the human visual system, they can replace clumsy serial logical inferences [13].

It is interesting to note that at the level of introductory programming, as argued by Lahtinen and Ahoniemi [25], most of the visualizations concentrate on *presenting* programming concepts, which facilitates mere understanding of the concept rather than appreciation of its application.

2.4 Name Conventions

The terms presented in this section, have not been formally defined. Here follows clarification of their use in this study.

End-user stance: the expression is used to denote the viewpoint of a non-expert towards a computing system, and it primarily addresses the predisposition towards a computer or a program as a distinct, irrelevant to one's concerns ontology.

Programmer stance: the term assumes awareness of one being *directly* involved in the computer operation processes.

Computing-expert stance: the term refers to the *mind-set* of an expert in a specific computing discipline.

Program Working Storage: the expression (also found as "program working memory") has been met in professional development contexts[5]. In general, it refers to the collection of addressable memory areas, while often it can also include implicitly used areas (such as index registers). The exact way, the term is used in this text is explained in detail in section 4.2.

[5] E.g. operating systems and database development.

3. PROGRAM-MEMORY INTERFACING

3.1 The Computer/Program Conception

As already mentioned, the disposition of the end-user is to concentrate on the task to be accomplished, and the "computer" or the "program" is there in order to serve this task. Although in practical terms both conceptions have the same usefulness for the end-user, it is necessary to stress a simple but important difference between them. While the "computer" view addresses a complete computer system, the "program" view assumes the existence of the computer and implies a distinction between them as two discrete entities that interact. Even further, usually as a result of attending a computer fundamentals course, where the basic computer operation is discussed, it seems that there is a rather fair understanding of the fact that programs are placed (loaded) in the main memory in order to be executed.

Whether the computer is using the program or vice versa, is not a question of this study. However, it is worth to note that experts distinguish between the memory area, where the program is stored, and the memory areas that the program *uses*. Under this perspective, given that a program is a programmer's specification, it is the programmer, who, through the program, is using the memory.

3.2 A Possible Threshold

To the author's knowledge, there has been no systematic research on the importance of a good understanding of program/memory interaction. It may be the case that it is so profoundly related to program comprehension and development that, although practiced and used during the course, it is *unconsciously* neglected. Evidence of its significance is additionally provided by the programming environments and the large number of approaches and empirically tested tools – designed either for teaching programming or for program comprehension (e.g. [14, 15, 34]) – which, as part of their functionality, expose the memory contents during program execution.

It seems that program/memory interfacing has a number of attributes that characterize a threshold concept. It can be thought as troublesome, for example, in the sense that while students may understand that a program is using memory, they do not realize *how* such use takes place, or their active role (through the program) in this process ("inert knowledge" [33]).

It can be assumed bounded since the term "memory" in real-life has a considerably different meaning (e.g. refers to both short-term and long-term memory). It may, also, be considered integrative in that it exposes an important part of the "hidden interrelatedness" of hardware with software, the programmer with the computer, etc.

Finally, it is transformative since, once understood, it will significantly shift the mentality of the learner, and, most probably, irreversible; according to Meyer and Land [32, 33], a concept, the acquisition of which leads to an "epiphany", is highly improbable to be forgotten or "unlearned".

4. THE PROGRAM WORKING STORAGE

The preceding discussion suggests that there is sufficient ground to *cultivate* the programmer's mentality, based on

evidently existing understandings of novice programmers. The aim is to cause the development of a mental model, which will be a) viable, so that it facilitates the learning of the programming practice and, therefore, the teaching objectives of an introductory programming course, b) valid, in that it will be consistent with the anticipated computing-expert mentality, and c) potentially *refinable*, in order to form the basis for more complex/specialized future models. To achieve that, the PWS is deployed as a conceptual model, which abstractly, nevertheless accurately, describes this part of the target system in which the programmer is actively involved.

4.1 Explanatory Basis

For better understanding of the ideas presented in the next section, assume a model, to be reflecting programs in general, using a) a *processing specification*, consisting of a set of statements, expressed in a certain notation and accessed in a predetermined manner, and b) a *data storage-repository*, formed by storage areas, accessed selectively as required by a statement.

Every statement expresses an instruction, which has a storage-repository related objective. In this respect, each statement is *accessing* the storage-repository in order to create a new storage area, use the data stored in one or more storage areas, modify the content of a storage area, or *release* a storage area.

Accessing can take place in one of the following ways:

a) Explicitly: the objective of the instruction involves a *specific storage area*, which can, in turn, be accessed in two possible ways:
   - Directly: referencing openly the interested storage area
   - Indirectly: referencing a lead to the interested storage area

b) Implicitly: the objective of the instruction involves *specific data*; the reference to the respective storage area is hidden.

The physical nature, the exact location, and the size of the storage areas, as well as, the implementation of stacks and heaps by the programming languages, are beyond the scope of the presented model. Storage areas, which, for convenience, will also be addressed as "memory areas", are only subject to the way they are accessed.

In the early stages of learning programming, memory constrains (e.g. stack size) and optimization techniques are not really an issue, primarily due to the fact that programs are small, algorithms are rather straightforward, and data structures are elementary.

4.2 The PWS as a Conceptual Model

4.2.1 Definition

In this study, *program working storage* is defined as a dynamic abstract entity, which is *program dependent* and, therefore, meaningful only in the context of a specific program or *program segment*. Program segment refers to a procedure, function, module, or an arbitrary selection of statement-sequence, with respect to the necessary declarations and initializations of the involved data-constructs (e.g. constants, variables, parameters).

The PWS is realized in two dimensions:

a) Space, as the collection of the short-term data storage areas (DASA) that are utilized by the program during execution.

b) Time, in terms of the states of DASAs throughout program execution.

A DASA is defined as an abstract unit, which corresponds to a physical short-term storage area, and has two primary characteristics: a) ability to hold a data value, and b) lifetime.

As data value is regarded any kind of value that can be stored, according to the semantics of the implementing programming language, independently of whether it can be explicitly manipulated. Usually[6], data values include characters, boolean and numeric values, and memory addresses.

As lifetime is regarded the time interval during which a physical storage area that corresponds to a DASA, is reserved by the program. DASAs with the shortest lifetime are usually implicitly accessed and hold literals and return-values. DASAs with the longest lifetime can be explicitly manipulated and represent global variables and constants.

4.2.2 Representation

The nature of the PWS, the students' limited understanding of the memory role, and the fact that the overall process is not evident impose the need of an external representation. Based on the research presented in sections 2.1 and 2.3, a suitable visualization can be justified as the best choice. Figure 1 depicts a simple representation of the PWS at a certain state.

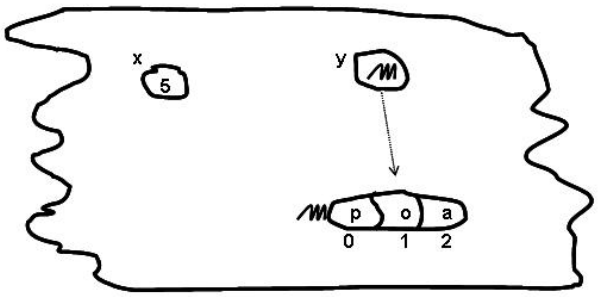

Figure 1

The minimalist form of the representation enables the learners to instantly produce states of a PWS even on simple media like paper, thus enhancing active involvement. On the other hand, teachers may enrich their representations with color, line formats, etc. (what Green and Blackwell have called "secondary notations" [7, 19]) in order to give emphasis or express additional information. Figure 2 depicts an instance of the PWS of the accompanying program. Colour is used to emphasize scope and dashed lines to indicate implicitly accessed DASAs.

[6] Considering the implementation languages used introductory programming courses, which are usually imperative or object-oriented.

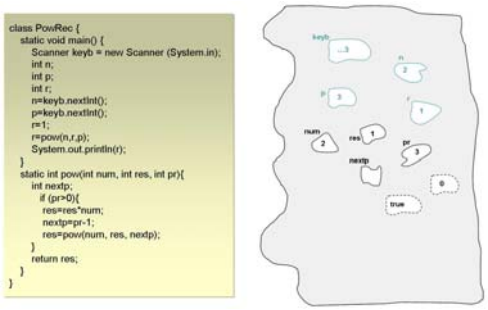

Figure 2

Even though the exact structure of the graphical representation is not a major concern, there are three recommendations:

- The storage-repository (memory at large) is arbitrarily shaped.
- DASAs are placed at relatively random positions.
- DASAs are shaped as to prevent misleading associations[7].

In any case, however, the goal is to draw attention to the significance of the "creation" and use of storage areas, rather than their physical nature.

4.3 The PWS as a Conceptual Framework

The PWS may be employed to present a variety of programming concepts and impinge their interrelatedness. At a basic level of use, it provides a means to identify and discuss memory/storage and data concepts, while parallel assimilation of its two dimensions (space and time) facilitates the study of essential programming constructs and problem-solving techniques.

Studied in space, the PWS can be used to emphasize explicit/implicit reference, as well as, direct/indirect manipulation, and, therefore, reveal major functionality features of variables, constants, parameters, data structures, literals, results of calculated expressions, and return-values. When examined in time, it can be very effective in highlighting scope of data constructs (presence and absence from states), and illustrating challenging attributes of data transition, such as copy and replacement of data-values.

Both dimensions of the PWS can be subject to *depth* in order to serve instruction as required (e.g. focus, student level of understanding). For space, depth refers to levels of abstraction of data structures, which can be suppressed, semi-suppressed, or expanded. For time, abstraction is related to *identifiable* program-execution, relative to memory-use, states. These may include segments, individual statements, or even expressions.

The study of the PWS in parts can also prove useful. Thorough examination of the states of selected DASAs may be used in order to underline the mechanics of control structures (e.g. iteration) and the roles of variables [24]. Selectively focused states may be used to stress the necessity of actions that will trigger the occurrence of the state in question, thus provoking program control flow and sequence. The acknowledgement of the concrete states of a DASA may facilitate the appreciation of the imposing action (statement or program segment), and, therefore contribute to the comprehension of the deployed algorithm.

It is worth to note that elementary optimization issues may also be addressed in terms of the DASAs' frequency of use.

[7] A common argument among educators is the inappropriateness of the use of a box to represent a variable.

5. DISCUSSION

The effectiveness of revealing the data-flow in accordance with control-flow becomes evident by the variety of approaches that utilize it. In their majority, tools that have been developed to assist the learning of programming include such a component. The PWS, however, attempts to take common practice one step further.

The PWS has been used in introductory programming courses for several years, with two different implementation languages – Pascal and Java. It occurred from the need to deal with novices' preconceptions – such as the conventional notion of memory and the mathematical view of variables – quickly and effectively.

Typical observations regarding the use of the PWS, which outline regular behaviour of approximately 300 students in 16 groups, include the following:

Using the graphical representation, students justify DASAs' requirements per line of code, while still early in the course. Moreover, it seems that they instantly appreciate the fact that *they* need to take actions, in order to acquire space for the data of the processing specification (variable/constant declarations).

Students avoid using areas, which in the depiction are blank. Eventually, they implement an initialization step at an appropriate position in the program, in order to produce the desired PWS state.

Using the graphical representation, students differentiate between DASAs which are directly or indirectly referenced, and are able to produce the *reference-path*. Eventually, data structures are identified as rooted directed graphs, the nodes of which are DASAs.

Students question actions (and, therefore, instructions) that produce DASAs which a) in the depiction remain blank throughout the PWS states, and b) after initialization are only accessed once. Progressively, they attempt to maintain just the necessary DASAs in every PWS state.

The frequency of use of the PWS is high:

- Early in the course
- When control structures are introduced
- When data structures are involved
- When sort and search techniques are introduced
- When recursive calls are involved
- During debugging exercises

Students tend to use the PWS less frequently towards the end of the term, as well as, when the processing specification and the data constructs are rather simple.

It is necessary to point that weak students tend to use the PWS more often than other students. Furthermore, it has been noted that weak students, who regularly use the PWS, consistently outperform those who do not and gradually reach a satisfactory level of understanding of "programming practice".

The fact that students use the depiction in order to justify their answers underpins the significance of the graphical representation. The fact that the explanations that the students provide are in accordance with the addressed programming rules is interpreted as evidence of the suitability of the PWS as a representational system. Its success is primarily attributed to: a) the ability to refer to the dynamic changes of memory usage in terms of time states, and b) the explicit demonstration of the relations between the memory and the program's instructions.

5.1 Assessing the Mental Model

The use of the PWS aims to trigger the programmer's mentality. Is the emerged mental model viable? Is it valid? Is it *refinable*?

At the end of the term, the average student is able to theoretically explain the taught concepts, comprehend how concepts are used in given problems, apply them to exercises, and proceed satisfactorily with a short project which requires analysis of components, synthesis and generalization of ideas presented in class, and evaluation of the possible implementation alternatives. Reflecting the educational objectives of Bloom's taxonomy [8], it seems safe to claim that the emerged mental model is consistent with the intended mindset an introductory programming course aims to grow.

In their study, Fix et al. [17] identify the following characteristics in experts' mental representations of programs: a) they are hierarchically structured, b) they have explicit mappings between the layers, c) they use basic recurring patterns for program comprehension, d) they are well-connected, and e) they are well-grounded, in that they "include specific details of where structures and operations physically occur in the program". Based on the presented observations, it seems that a mental representation, generated by a user of the PWS, will be relatively consistent with the anticipated expert mentality, since a) the PWS is by nature hierarchical, b) its study requires explicit mappings between layers, c) it facilitates the detection of recurring patterns, d) it enforces the appreciation of components' interrelatedness, and e) every state, usually provides enough details to locate a certain operation in the program.

Finally, monitoring the progress of the students in the "Computer System Architecture" course, it proved they were able to efficiently reposition memory usage at the appropriate "hardware" locations, based on the attributes of implicitly vs. explicitly referenced storage areas.

6. CONCLUSION

Being abstract enough to fit the semantics of potentially any programming language, the PWS methodology may be integrated in the teaching/learning process of introductory programming, as a means to a) present several programming concepts, while revealing their interrelatedness, and b) address crucial pre-conceptions of novices.

A weak point of the system's current implementation media (traditional media, such as pen & paper) is that reproducing the PWS states may be quite "space-consuming" and, eventually, depending on the program's complexity, inefficient. Program animation tools, like Jeliot [34], may be productively used (to a certain extent) for support. Nevertheless, the development of tools, which will enable the users to *design* the PWS states and, thus, facilitate the major learning dimension of the methodology, is almost compulsory.

How constrained the representation should be, the exact implementation in the context of various programming languages, the level at which the emerged mental model suffices for the programming practice, and the ways it can be refined are some of the research questions that arise from this study.

The PWS is not, by any means, presented as a panacea for refining all possible preconceptions or dealing with all difficult concepts, like an introductory programming course does not aim to develop computing experts. The necessity of a viable computer model in the early stages of the study of any computing discipline is acknowledged in this thesis. However, due to facts and constrains presented in section 1, the suggestion is that its development takes place in the time/space frame of all introductory courses.

7. ACKNOWLEDGEMENTS

Many thanks to J. Kiourktsoglou, R. Cox, and R. Lutz for their valuable comments.

8. REFERENCES

[1] ACM/AIS/IEEE-Curriculum 2005 Task Force. *Computing Curricula 2005*. IEEE Computer Society Press and ACM Press, September 2005. (http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf)

[2] ACM/IEEE-Curriculum 2001 Task Force. *Computing Curricula 2001, Computer Science*. IEEE Computer Society Press and ACM Press, December 2001. (http://www.acm.org/education/cc2001/final/index.html)

[3] Ben-Ari, M. Bricolage Forever! In *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group*, University of Leeds, UK, 1999.

[4] Ben-Ari, M. Constructivism in Computer Science Education. In *Proceedings of the 29th SIGSCE Symposium*, Atlanta, USA, February 1998.

[5] Ben-Ari, M. *Understanding Programming Languages*. John Wiley & Sons, 1996. (http://stwww.weizmann.ac.il/G-CS/BENARI/books/)

[6] Blackwell, A.F. First Steps in Programming: A Rationale for Attention Investment Models. In *Proceedings of the IEEE Symposia of Human-Centric Computing Languages and Environments*, pp. 2-10, 2002.

[7] Blackwell, A.F., Green, T.R.G. Investment of Attention as an Analytic Approach to Cognitive Dimensions. In T. Green, R. Abdullah & P. Brna (Eds.), *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pp. 24-35, 1999.

[8] Bloom, B.S. *Taxonomy of Educational Objectives: The Classification of Educational Goals – Handbook 1: Cognitive Domain*. Longmans, 1965.

[9] Bonar, J. & Soloway, E. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1(2), pp. 133-161, 1985.

[10] Boudreau, G. & Pigeau, R. The Mental Representation and Processes of Spatial Deductive Reasoning with Diagrams and Sentences. *International Journal of Psychology*, 36(1), pp. 42-52, 2001.

[11] Braine, M.D.S. & O'Brien, D.P. A Theory of It: A Lexical Entry, Reasoning Program, and Pragmatic Principles. *Psychological Review*, 98, pp. 182-203, 1991.

[12] Braine, M.D.S. On the Relation between the Natural Logic of Reasoning and Standard Logic. *Psychological Review*, 85, pp.1-21, 1978.

[13] Cheng, P.C.-H. Unlocking Conceptual Learning in Mathematics and Science with Effective Representational Systems. *Computers in Education*, 33(2-3), pp. 109-130, 1999.

[14] DuBoulay, B. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), pp 57-73, 1986.

[15] Efopoulos, V., Dagdilelis, V., Evangelidis, G. Satratzemi, and M. WIPE: A Programming Environment for Novices. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Caparica, Portugal, 2005.

[16] Etmer, P.A. and Newby, T.J. Behaviorism, Cognitivism, Constructivism: Comparing Critical Features from an Instructional Perspective. *Performance Improvement Quarterly*, 6(4), pp. 50-70, 1993.

[17] Fix, V., Wiedenbeck, S, Scholtz, J. Mental Representations of Programs by Novices and Experts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, 1993.

[18] Gonzalez, G. Constructivism in an Introduction to Programming Computer Course. *Journal of Computing Science in Colleges*, 19(4), pp. 299-305, 2004.

[19] Green, T. Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities. In *Proceedings of the Working Conference of Advanced Visual Interfaces (AVI2000)*, Palermo, Italy, 2000.

[20] Haberman, B. & Kolikant, Y.B.D. Activating "Black Boxes" instead of Opening "Zippers" – a Method of Teaching Novices Basic CS Concepts. In *Proceedings of ITICSE 2001*, Canterbury, UK, pp. 41-44, 2001.

[21] Johnson-Laird, P.N. & Byrne, R.M.J. Precis of Deduction. *Behavioural and Brain Sciences*, 16, pp. 323-380, 1993.

[22] Johnson-Laird, P.N. *Mental Models: Towards a Cognitive Science of Language, Inference, and Conciousness*. Cambridge University Press, Cambridge, 1983.

[23] Jonassen, D.H. Objectivism versus Constructivism: Do We Need a New Philosophical Paradigm? *Educational Technology Research and Development*, 39(3), pp.5-14, 1991.

[24] Kuittinen, M. and Sajaniemi, J. Teaching Roles of Variables in Elementary Programming Courses. In *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, Leeds, UK, 2004.

[25] Lahtinen, E. & Ahoniemi, T. Visualizations to Support Programming on Different Levels of Cognitive Development. In *Proceedings of the 5th Koli Calling Conference on Computer Science Education*, Koli, Finland, November 17-20, 2005.

[26] Larkin, J.H., Simon, H.A. Why a Diagram Is (Sometimes) Worth a Thousand Words. *Cognitive Science*, 11, pp. 65-100, 1987.

[27] Lui, A.K., Kwan, R., Poon, M., Cheung, Y.H.Y. Saving Weak Programming Students: Applying Constructivism in a First Programming Course. *ACM SIGSCE Bulletin*, 36(2), pp. 72-76, 2004.

[28] Mavaddat, F. An Experiment in Teaching Programming Languages. *ACM SIGCSE Bulletin*, 8(2), pp. 45-59, 1976.

[29] Mayer, R.E. Different Problem-Solving Competencies established in Learning Computer Programming with and without Meaningful Models. *Journal of Educational Psychology*, 67, pp. 725-734, 1975.

[30] Mayers, B.A. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1, pp. 97-123, 1990.

[31] McCartney, R. and Sanders, K. What are the "Threshold Concepts" in Computer Science? In *Proceedings of the 5th Koli Calling Conference on Computer Science Education*, Koli, Finland, November 17-20, 2005.

[32] Meyer, J. & Land, R. Threshold Concepts and Troublesome Knowledge (2): Epistemological Considerations and a Conceptual Framework for Teaching and Learning. *Higher Education*, 49(3), pp. 725-734, 2005.

[33] Meyer, J. and Land, R. Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within Disciplines. *ETL Project Occasional Report 4*, Universities of Edinburgh, Coventry, and Durham, 2003.

[34] Moreno, A. & Myller, N. Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In *Proceedings of International Conference on Networked E-Learning for European Univarsities*, Granada, Spain, 2003.

[35] Pane, J.F. & Myers, B.A. Usability Issues in the Design of Novice Programming Systems. *Technical Report CMU-CS-96-132*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, USA, 1996.

[36] Pea, R.D. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, 2(11), pp. 25-36, 1986.

[37] Petre, M., Baecker, R, & Small, I. An Introduction to Software Visualization. In J. Stasko, J. Domingue, M.H. Brown, B.A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience*, MIT Press, pp. 3-26, 1998.

[38] Powers, K.D. Teaching Computer Architecture in Introductory Computing: Why? and How? In *Proceedings of the 6th Australasian Computing Education Conference (ACE2004)*, Dunedin, New Zealand, January 2004.

[39] Shu, N.C. Visual Programming: Perspectives and Approaches. *IBM Systems Journal*, 28(4), pp. 11-34, 1989 (reprinted 1999).

[40] Smith, J.P., diSessa, A.A., Roschelle, J. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The Journal of the Learning Sciences*, 3(2), pp. 115-163, 1993.

[41] Viner, M. Constructivism and Educational Implications for Teaching and Learning. *Journal of Educational Computing, Design & Telecommunications*, 3(1), 2003.

[42] VonGlasersfeld, E. A Constructivist Approach to Teaching. In Steffe and Gale (Eds.), *Constructivism in Education*, Hillsdale, NJ, Lawrence Elbaum Associates, pp. 3-15, 1995.

[43] White, G. Misconceptions in CIS Education. *Journal of Computing Sciences in Colleges*, 16(3), 2001.

[44] Wulf, T. Constructivist Approaches for Teaching Computer Programming. In *Proceedings of the 6th Conference on Information Technology Education (SIGITE '05)*, Newark, NJ, USA, 2005.