Learning Programming Using Program Visualization Techniques

Lynne P Baldwin and Jasna Kuljis
The VIVID Centre
Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex UB8 3PH, UK
Email: {Lynne.Baldwin, Jasna.Kuljis}@brunel.ac.uk

Abstract

This paper describes the programming knowledge and skills that learners need to develop, and concludes that this is an area of computer science education where those involved in the teaching of programming need to further explore the nature, structure and function of domain-specific knowledge. It has been argued that conceptual models can serve to enhance learners' conceptual understanding of programming. The methods used to enhance the development of accurate mental models include: designing the interface so that users can interact actively with it; using metaphors and analogies to explain concepts; and using spatial relationships so that users can develop capabilities for mental simulations. With the insights afforded by looking more closely at conceptual understanding, we describe how visualization techniques may effectively be used in the learning and teaching of programming.

1. Introduction

Since the introduction of personal computers and, later, unrestricted access to the Internet and the World Wide Web (www) the computer is becoming one of the most ubiquitous tools in society today, both for those at work and at home. However, the only way most people use computers is through application programs. Most people do not realise that computers are much more flexible and much more powerful tools than say merely something to be used for the writing of letters. A powerful analogy might be drawn between the pianola and the piano; on the pianola we can play only pre-recorded music, whereas the piano allows us to both to play any music and also to compose our own. It would seem that the view of the computer held by those learning programming in institutions of higher education and elsewhere today view the computer much as a pianola; that is, they see the computer as only a tool, or an instrument, that allows them to access the Internet, do some simple data processing tasks, or to do some word processing. For those teaching programming, the task is made all the more difficult as it involves helping learners to change their view of the computer from that of a pianola to that of a piano. That is to say, that the computer is a tool that can be programmed in an infinite number of ways, limited only by the skill and creativity of the programmer.

The majority of students, even those enrolled on computer science courses, find computer programming a difficult and complex cognitive task ([10], [12], [15], [17], [28], and [20]). It is therefore not surprising to find that it is a topic that generates much discussion among those teaching in higher education in various educational settings across the globe. One such debate, involving British academics (cphc-members@mailbase.ac.uk) but is arguably doubtless similar to those elsewhere, reported their experience in using various teaching methods in order to improve their teaching of programming, and thus the pass rates in their classes. Changing the language that is taught first does not significantly change the pass rate. Neither does using different textbooks, slowing the course down, nor alternating between a bottom-up or a top-down approach. Some academics opted to reduce the quantity of material taught and concentrate on what they considered the most important topics, although some expressed the view that this served only to lower standards. Because of the enormous increase in the student population, in the UK at least, the classes are large; 100-200 students in computer programming courses is not uncommon. As only to be expected in any class, and with the added challenge of larger numbers of students in them, there is a huge variety in students' abilities, learning speeds, and attitudes. This makes it all the more difficult for the teacher to devote time and attention to individuals who may be finding the learning of programming problematic.

A not-unrelated issue is that the majority of students consider courses in programming an unnecessary evil, but that it is one that ends, thankfully, with graduation. Final year students often cannot program at all according to Smith et al. (2000), who report the estimate made by Elliot Soloway, Director of the Highly Interactive Computing project at the University of Michigan, namely, that less than 1% of novice programmers continue to program after their programming class ends. Therefore, educators are faced with the task of having to find alternative, or additional, ways of helping students in their learning by providing a variety of modes of learning which support different rates of learning. One possibility is to let students teach themselves with the help of computer. However, this raises the challenge for those designing the learning activities for such learners, particularly in light of the comment that teachers of programming have long been criticized for failing to develop students' understanding of programming in the key area of program comprehension ([11], [18], [23], and [24]).

Learners have been appropriately guided it seems, by way of formal instruction in the classroom, in acquiring the 'how to' skills of programming, but have not been assisted to understand the 'why'. Designing appropriate courseware, whether for the classroom or for a computer-based learning package is, then, an added challenge for those teaching programming, as it appears that we still have much to discover about not only programming itself, but of both learning and teaching it. Clear [3] suggests that those involved in the teaching of programming need to reconsider their approach to teaching, and that current theories on cognition may require the adoption of a more inductive, exploratory and interactive approach; a move away from seeing programming as "a process of detached, abstract reflection and consideration" to one of "active engagement and action" (p. 25). He comments that abstraction followed by action may not be valid planning or programming techniques, and posits that models of learning which emphasize interaction may be more powerful. Visualization techniques, posits Clear [3], may offer important insights into the learning and teaching of programming. If, as many of those teaching it seem to agree, programming is so difficult, the following section addresses the possible reasons for this.

2. Challenges Associated with Learning Programming

Programming is a cognitively challenging task, and one which involves many skills. McGill and Volet [21] comment that it is now generally accepted that learners need to acquire and use three interrelated types of knowledge of programming. One, syntactic; two, conceptual; three, strategic. Syntactic knowledge is knowledge about specific facts about a programming language, and its rules for use. Learners thus need to acquire this technical knowledge in order to write programs which compile, although at this stage they are unable to design and develop programs which solve problems. Conceptual knowledge concerns constructs and principles of computer programming, and for this learners need to develop mental models of the system, and the semantics of the program actions. With both syntactic and conceptual knowledge, learners are able to design solutions to simple, or closely-related, problems that they have met in the classroom. Strategic knowledge concerns general problem-solving skills which are programming-specific, and with syntactic, conceptual and strategic knowledge learners are able to solve novel programming problems. It is strategic knowledge that is needed for the recognition and decomposition of a problem, as well as for testing and debugging errors, as well as designing the phases of programming.

Although it might appear that these are separate areas of knowledge, and that as such they might be learned incrementally, learning is naturally more complex than this. Our current knowledge of how learners learn to program is very limited, but however it is organized or described, the ability to create generalizations and abstractions is central. A programmer needs to design a program architecture, break large or complex tasks into smaller or simpler and more manageable tasks, and also to choose appropriate data structures and algorithms. One of the difficulties in doing these tasks is that learners have little, if any, previous experience to draw on in order to help them, although the study of mathematics may, perhaps, be useful in this regard. The main problem that novice programmers have when programming computers is the gap between the representations the brain uses when thinking about a problem and the representations a computer will accept. Norman [22] argues that there are only two ways to bridge that gap: one, move the user closer to the system; two, move the system closer to the user. Programming classes try to do the former, that is, move the user closer to the system, by focusing on teaching students a programming language.

The learning and teaching of programming is not, naturally, isolated from learning and teaching more generally, and any discussion thus needs to take account of this. To this end, the following section looks at learning and teaching more generally, and how this informs the teaching of programming. We draw attention to the role of verbal and visual metaphors and analogies that are used in the various approaches to teaching programming and then describe how visual support might aid learners in the learning of programming.

3. Programming within the Wider Context of Learning and Teaching

Any discussion on the learning and teaching of programming naturally needs to make reference to the theories that concern learning and teaching more generally, as these are inextricably entwined. Together with specific knowledge about programming itself, learning programming demands complex cognitive skills such as planning, reasoning and problem-solving, and there is currently much effort devoted to trying to discover which higher level thinking skills learners need for the task ([2], [26], and [34]). These skills are, however, not domain-specific, and although general intellectual ability, especially logical reasoning and spatial ability play their part in learning how to program, we know little about either what such complex intellectual skills are, or how they are used.

Shih and Alessi [27] comment that theories which attempt to account for how learners develop conceptual understanding appear to offer differing explanations of the mechanisms which underlie such learning. The notion that beginners start at a 'low' level and gradually 'move up' in some way is common, and 'mastery', or 'expert-level' expertise is often used to describe the learning process. The view of learning which underlies the literature in computer science education suggests that knowledge can be neatly parceled up into chunks for both learning, and teaching, purposes, and this has an important impact on the design of material for the classroom and also for how this should be delivered. It suggests that teachers of programming need only present material in what might be considered a 'logical' order, and that if it is well presented, learners will somehow 'absorb' this, and will then be able to be presented with the next seemingly 'logical' concept.

This view of teaching in Higher Education, which Gibbs [8] calls one of 'disseminating knowledge' sees knowledge as something that exists separately from those who possess it, and also as something that can be conveyed; theory and practice are separate domains. The academic and social environment is seen as of little importance, as knowledge is created through a social system but learned through individual study and practice. For the teachers who share this view, their role is to ensure that content is covered, and that presentation and organization of knowledge is effectively delivered. Students are rewarded for successful completion of the tasks given to them. This is a model of teaching that is, arguably, still much in evidence today yet, as we will now discuss in part, does not take account of newer theories about knowledge, and thus learning.

Although little is known about how we process what we see, hear, touch, taste or smell, Gibbs [8] suggests a newer, but contrasting model, namely that of 'making learning possible'. This draws on a later theory of learning than the one which underpins 'disseminating knowledge', that is, of individual and social constructivism [19]. For the teacher who shares this later view, knowledge does not exist separately from those who possess it, and is instead something that is reconstructed by learners. Effective learning thus takes place in an environment where learners have the opportunity to interact and cooperate; essential for negotiating (rather than 'absorbing') meaning. In terms of the role of the teacher, content is restricted to the barest essentials, and tasks for students are designed with learning outcomes very much in mind. Dialogue with learners is key, particularly as this affords the opportunity to challenge the perhaps erroneous beliefs that learners hold about concepts. Although teachers who hold this view of teaching want their learners to learn, success is not seen as the successful completion of tasks but is instead concerned with how learners change and develop, and ensuring that the learning environment is designed to enable this. Also central to this model of teaching is that concepts and principles are abstracted from experience. This is in stark contrast to the 'disseminating knowledge' model whereby theory is presented by the teacher (or book, perhaps) and practice is left to learners to do, often with little guidance or dialogue.

It is unsurprising, in the UK at least, that teachers of programming may not have embraced this newer model of teaching as they have, for the most part, not been trained to teach before setting foot in the classroom and thus not only have to draw on their past experiences of the classroom to guide them but also have not had the opportunity to review current theories on learning and teaching.

The design and use of visual systems naturally focuses on practical experience, and with it the opportunity for learners to abstract the concepts and principles of programming. Interaction is thus emphasized, as well as the role of the learner. This is, then, much more in line with the above newer model of teaching, and is a welcome first step in finding out more about how learners learn this complex skill. However, whatever the model of teaching that teachers of programming hold, delivery of material often involves an oral explanation of concepts, often accompanied by visual material, and by its nature involves using language. Metaphor and analogy are devices common to all natural languages, and it therefore follows that our use of these in everyday communication finds its way into the language that we use in our classrooms. However, although we use natural language as one (of many) means of communication at our disposal, interacting with others in this way involves constant re-

negotiation of meaning if we are to reach some shared understanding. Analogy and metaphor are, then, only useful in aiding that understanding if those involved in the dialogue share a similar (it can never be the same) view of what they represent. The challenge for us, as educators, is to explore which metaphors and analogies might prove useful, and also when and how they might be used in using visual systems. This is further explored in the following section, since it is the backbone of our research. Other theories, such as structuration theory [9] and its applications to organizational learning ([25], and [5]) clearly exist, but are not considered here at this point in our research.

4. Role of Metaphors and Analogies in Learning

Metaphors and analogies are often used to explain complex concepts or to illustrate processes. Those teaching programming often support their classroom activities with various verbal or visual metaphors. Visual metaphors are arguably much more powerful than verbal ones, and thus computer science books are often filled with illustrations. Visual representations are used to illustrate, among others, data structures, algorithms and problem decomposition. Examples of such metaphors are presented in Figures 1 and 2:

Even though we use these visual metaphors with the aim of helping learners understand problems, they are then required to perform cognitive transformation; from the visual representation to actual program code. It would be much less demanding if they could communicate to the computer using the same visual forms as they do when they analyze a problem.

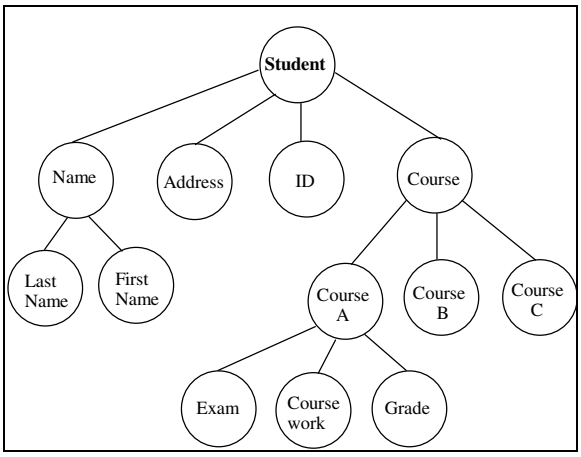

Figure 1. An example of a tree data structure

Lakoff and Johnson [16] describe metaphors as "understanding and experiencing one kind of thing in terms of another", and they claim that metaphors are not only pervasive in language, but that they are a fundamental part of our conceptual system of thought and action. Metaphors are such an integral part of our language and thought that they are almost invisible; we use them without consciously being aware of doing so. Erickson [7] describes a metaphor as an invisible webs of terms and associations that underline the way we speak and think about concepts. Metaphors function as natural models, allowing us to take our knowledge of familiar, concrete objects and experiences and use it to give structure to more abstract concepts. Therefore, when generating metaphors, we have to be aware that metaphors are already implicit in the problem description and in the description of functionality of abstract concepts [7].

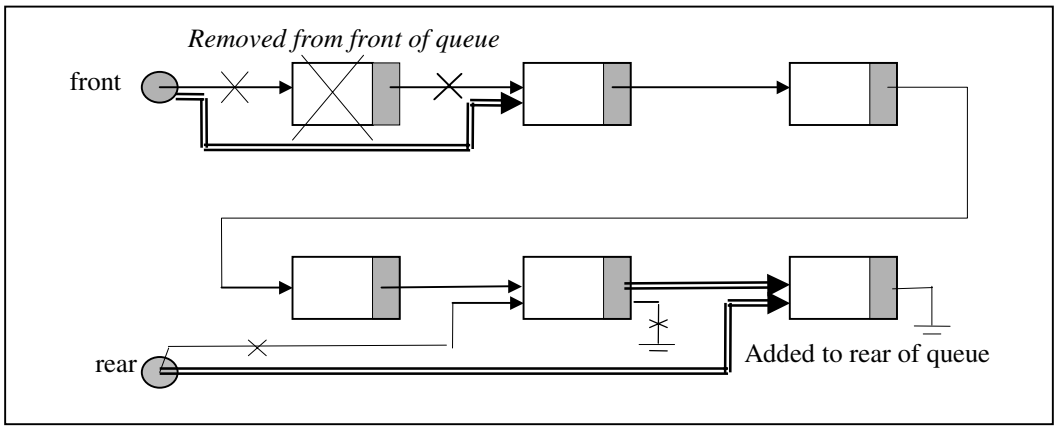

Figure 2. Moving a node from the front to the rear of a queue

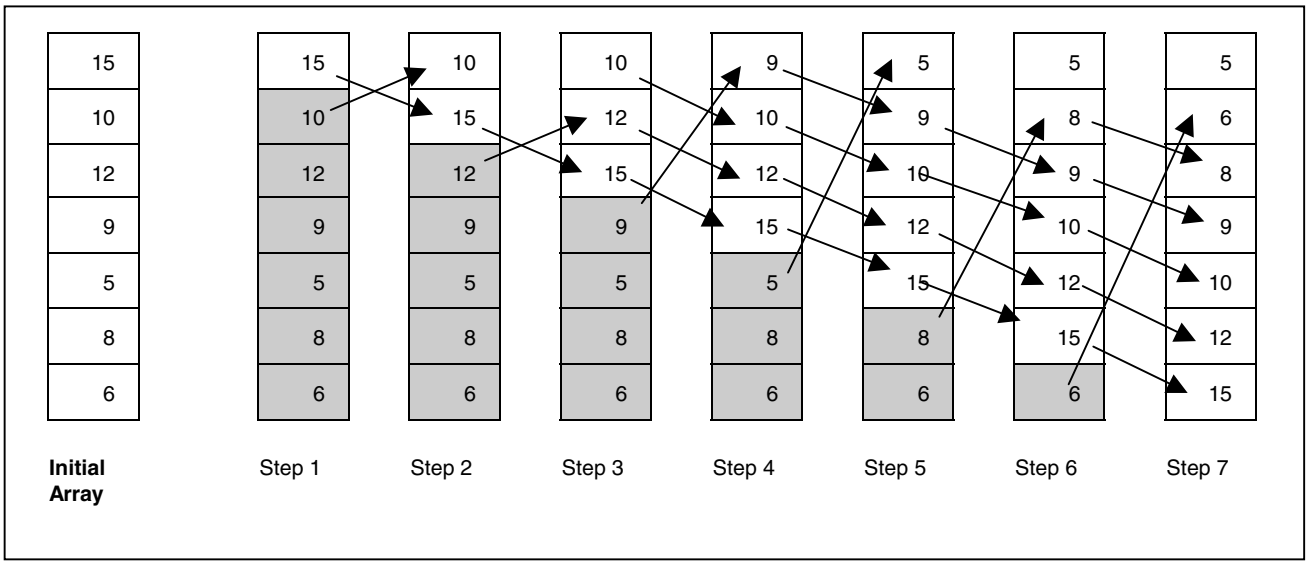

Figure 3. Insertion sort performed on six integers stored in an array

A well chosen interface metaphor therefore offers the opportunity to provide users with realistic expectations about what will happen. It is, however, vital that whatever metaphor, or representation, is chosen, it is sufficiently familiar to learners; if not, it follows that they will not be aided in their learning. Sloman [29] divided representations into two general types: analogical and 'Fregean'. In analogical representation the structure of the representation gives information about the structure being represented. In the Fregean system (named after Gottlob Frege, the inventor of predicate calculus) there is only one type of expressive relation between parts of a configuration, namely, the relation between function signs and argument signs. The structure of such a configuration need not correspond to the structure of what it represents or denotes. Most programming languages use Fregean representations, which aim to be both general and powerful.

Analogy allows for flexibility in applying existing knowledge to new situations. The standard computational model of analogy defines the source of analogy to be a problem solution, example, or theory that is relatively well understood, although the target may well not be completely understood. Analogy constructs a mapping between the corresponding elements of the target and its source. In considering the possible representations of a problem, it is useful to adopt the three classifications of representations put forward by Bruner [1], namely, enactive, iconic, and symbolic. Enactive representation employs a set of actions appropriate for achieving a certain result. Iconic representation employs a set of summary images or graphics that stand for a concept without defining it fully. Symbolic representation uses a set of symbolic or logical propositions drawn from a symbolic system that is governed by rules or laws for forming and transforming propositions. In Bruner's classification, enactive and iconic representations are analogical representations whereas symbolic representation is Fregean.

The use of visual languages appear to offer advantages over more 'traditional' ones in helping learners to learn programming more easily, and the following section describes both the variety of visual languages currently available and their perceived benefits.

5. Visual Support to Learning Programming

There are many varieties of visual languages, among them Stagecast Creator and ToonTalk, which are described in greater detail later in this section. Some languages work almost exclusively through graphical elements where virtually every aspect of the language, including numerical operations, has been replaced by some sort of graphical representation. Some mix textual and iconic elements, and others use the visual element to affect program layout, using flow diagrams or nested boxes to lend structure to basically textual programs [6].

There is not enough evidence to lead us to conclude that visual languages are easier to learn than other styles of languages, however tempting it might be. Intuition leads us to suspect, however, that visual languages must somehow be 'better' than conventional ones. We are not alone in our belief as there are still researchers pursuing the goal of making learning easier or more fun through visual programming; in particular, in programming by demonstration (PBD) and the end-user programming

communities. The main goal of PBD is to shield the user from the need to learn such programming concepts as variables, loops, and conditionals [4]. In such systems users do not need to learn programming syntax, they have to learn only very little beyond the informal operations that they already know how to perform with software. However, there are problems in expressing complex tasks and it seems that users often have difficulty in providing unambiguous examples of a more general procedure [6].

Figure 4..Stegecast: Sokoban [33]

Stagecast Creator (Smith et al., 2000) is a visual system for novice programmers. It is the product of seven years of research on a project initially called KidSim [31] and which was later renamed Cocoa [30] before becoming known as Creator. The goal of the Creator project is to empower end users, both teachers and learners, to construct and modify simulations through programming. The Creator Project [32] uses a new approach, one which eliminates traditional programming languages in favor of a combination of two technologies: PBD and visual before-after rules. In PBD, users demonstrate algorithms to the computer by operating the computer interface just as they would if they were not programming. The computer records the user's actions and can then re-execute them later on different inputs. Creator uses analogical representations in its rules and allows these representations to be directly manipulated in the process of programming (see Figure 4). The programmer specifies the behavior using graphical rewrite rules.

Learning by demonstration, an approach that has been successfully used in many fields of learning in both formal and informal contexts, has similarly been applied in the field of learning programming. Programming by demonstration thus allows learners to directly manipulate the representations, and graphical rewrite rules provide an understandable representation for the recorded programs. This approach does not, however, necessarily lead to learning programming. There is some empirical evidence (Smith et al., 2000) which indicates that programming using Creator is easier for novices than using a programming language syntax. However, Creator addresses only the specialized domain of visual simulations and is aimed at a particular kind of learner. KidSim, as the original name for Creator suggests, was initially designed to enable young learners to learn programming.

Another similar PBD system, again designed for children, is ToonTalk ([13], and [14]). The approach to programming used here is to eliminate programming language syntax and to provide animated programming. The programmer is a character in an animated virtual world in which programming abstractions are replaced by their tangible analogs. ToonTalk includes several narrated demonstrations, a puzzle sequence consisting of about 70 puzzles, and an interactive coach guide character (see Figure 5). Everything in ToonTalk can be seen, picked up, and manipulated. Children can build and run programs by doing thing like giving messages to birds, training robots to work on boxes, loading up trucks, and using animated tools to copy, remove and stretch items.

Figure 5. ToonTalk: Here the user has picked up a bird and Marty is talking about what birds do [14]

Whether young or old, the similarity between both types of learner is that they are new to programming and, by implication, if children can learn, it follows that it must be similarly easy (or indeed eas*ier*, perhaps) for adult learners too. This presupposes that children and adults learn in similar ways, and can thus be taught similarly which may, or may not, be the case. Nevertheless, the

above developments are a welcome first step, particularly in light of how little is known about the learning and teaching of programming.

There are some not inconsiderable differences that need to be taken into account when designing such a system for adults, not least of which is that children see what they are doing on the computer as fun; the same is certainly not true for adults, or at least, not for the majority of those who learn programming as part of their degree course. Children, unlike learners in Higher Education, do not have a limited timescale to learn programming, nor are they assessed at regular intervals and, importantly, they do not have to write code. Children thus have few, if any, of the pressures, responsibilities or anxieties of the adult learner, which makes designing programming courses all the more problematic for the teacher of programming in Higher Education.

For too many learners who start a degree in the discipline of computer science or information systems, the fear surrounding programming is palpable. Although programming is undeniably complex, it raises the interesting question as to why it is that programming, rather than any other subject or topic, that engenders such fear. All new activities, including programming, are naturally a little unnerving for the newcomer. The task of the teacher of programming is, in part, to both help learners in acquiring the skills needed for programming but also to increase their confidence in, and enthusiasm for, programming. This is made all the more difficult given that this can be eroded by many other factors that complicate the issue. For the student in the first semester of their first year, these include having to adjust to not only a new learning environment, but also (perhaps) a new home, to living more independently, managing (very limited) finances for the first time and, increasingly, fitting studies around part-time (in some cases, full-time) work. It is not unsurprising that students are a little anxious, and that this extends into the classroom.

Given these factors, and the fact that many are new to programming, designing appropriate teaching and learning activities needs to be tailored in order to take sufficient account of learners' lack of confidence at this particular time in their degree course. While making programming 'fun' is doubtless an unrealistic goal for adult learners in Higher Education, systems such as Creator and ToonTalk suggest that a closer look at visual systems may offer opportunities that better aid learners at this critical period in their development of programming skills.

6. Implications and future work

There is still much research needed in order that we can better support learning in general and learning programming in particular. We are currently conducting experiments which evaluate not only which visual objects but also which representational paradigms best support learners in their learning of programming. In graphical programming, various diagramming techniques have been used as program specification tools. A flowchart, for example, represents a diagram assembled of graphical objects representing an algorithm of a program. Since its introduction, many other diagramming techniques have evolved, and all of them are said to aid programming. Given that these have met with limited success, it suggests that there is still much to be done in order to develop a visual environment that better supports learners in their learning. The work mentioned in this paper, as stated earlier, is based on a metaphor and analogies applied to learning. We recognize that other theories might prove in the long run more beneficial ([9], [5], and [25]).

7. References

[1] J. Bruner, Theory of Instruction. Harvard University Press, Cambridge, Mass., 1966.

[2] R. E. Clark and S. B. Blake, Designing training for novel problem-solving transfer. In Tennyson R. D., Schott F., Seel N. M. and Dijkstra S. (eds) *Instructional Design: International Perspectives Volume 1: Theory, Research and Models*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996, pp. 183-214.

[3] T. Clear, The nature of cognition and action, *ACM SIGCSE Bulletin*, Vol. 29, No.4, 1997, pp. 25-29.

[4] A. Cypher, Introduction: Bringing programming to end users. In A. Cypher (ed.), *Watch What I Do*, MIT Press, Cambridge, MA, 1993, pp. 1-11.

[5] G. DeSanctis and M. S. Poole, Capturing the complexity of in advanced technology using adaptive structuration theory, *Organization Science*, Vol. 5, No. 2, 1994, pp. 121-147.

[6] M. Eisenberg, End-user programming, In G. Helander, T.K. Landauer, P. V. Prabhu (eds.) *Handbook of Human-Computer Interaction*, Elsevier Science, Amsterdam, 1997, pp. 1127-1146.

[7] T.D. Erickson, Working with Interface Metaphors, In R.M. Baecker, J. Grudin, W.A.S. Buxton, and S. Greenberg (Eds.) *Human-Computer Interaction: Toward the Year 2000*, Second Edition, Morgan Kaufmann, San Francisco, 1995, pp.147-151.

[8] G. Gibbs, *Improving Student Learning*. The Oxford Centre for Staff Development, Oxford, UK. 1994.

[9] A. Giddens, *The Constitution of Society*, Polity Press, Cambridge, UK, 1984.

[10] R. Guindon, Designing the design process: exploiting opportunistic thoughts. *Human Computer Interaction*. Vol. 5, 1990, pp. 305-344.

[11] C. T. Haynes, Experience with an analytic approach to teaching programming languages. Proceedings of *the 29th SIGCSE Technical Symposium on Computer Science Education*, (25 February - 1 March 1998, Atlanta, Georgia), *ACM SIGCSE* 30, 1, 1998, pp. 350-354.

[12] R.A. Jeffries, P. Turner, G. Polson and M.E. Atwood, The processes involved in designing software. In J.R. Anderson (Ed), *Cognitive Skills and their Acquisition*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1981, pp. 255-283.

[13] K. Kahn, Generalizing by removing detail, *Communications of the ACM*, Vol. 43, No. 3, March 2000, pp. 104-106.
[14] K.Kahn, ToonTalk Home Page <http://www.toontalk.com> Animated Programs.
[15] J. Kim and F.J. Lerch, Why is programming (sometimes) so difficult? Programming as scientific discovery in multiple problem spaces. *Information Systems Research*, Vol. 8, No. 1, 1997, pp. 25-50.
[16] G. Lakoff and M. Johnson, *Metaphors we live by*. University of Chicago Press, Chicago, IL, 1980.
[17] S. Letovsky, Cognitive processes in program comprehension. In E. Solway and S. Iyengar (Eds). *Empirical Studies of Programmers*. Ablex Publishing, Norwood, NJ, 1986, pp. 58-79.
[18] M. C. Linn and M. J. Clancy, Can experts' explanations help students develop programming design skills? *International Journal of Man-Machine Studies*, 1992.
[19] F. Marton and S. Booth, *Learning and Awareness*. Lawrence Erlbaum Associates, Mahwah, NJ, 1997.
[20] R.E. Mayer, The psychology of how novices learn programming. In E. Soloway and J.C. Spohrer (Eds). *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989, pp. 129-159.
[21] T. J McGill. and S. E. Volet, A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education*, Vol. 29, No. 3, 1997, pp. 276-297.
[22] D. Norman, Cognitive engineering, In D. Norman and S. Draper (eds.) *User Centered System Design*, New Perspectives on Human-Computer Interaction, Lawrence Erlbaum, Hillsdale, N.J., 1986, pp.31-61.
[23] R. Oliver, Measuring hierarchical levels of programming knowledge. *Journal of Educational Computing Research,* Vol. 9, No. 3, 1993, pp. 299-312.
[24] R. Oliver and J. Malone, The influence of instruction and activity on the development of semantic and programming knowledge. *Journal of Research on Computing in Education,* Vol. 25, No. 4, 1998, pp. 521-533.
[25] W. J. Orlikowski, The duality of technology: rethinking the concept of technology in organizations, *Organization Science*, Vol. 3, No. 3, 1992, pp. 398-427.
[26] S. Robins and R. E. Mayer, Schema training in analogical reasoning. *Journal of Educational Psychology,* Vol. 85, No. 3, 1993, pp. 529-538.
[27] Shih Y. and S. Alessi, Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education,* Vol. 26, No. 2, 1994, pp. 154-175.
[28] H.A. Simon. The structure of ill-structured problems. *Artificial Intelligence*, Vol. 4, 1973, pp. 181-201.
[29] A. Sloman, Interaction between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence, In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence* (London), Morgan Kaufman, San Francisco, 1971, pp. 270-278.
[30] D.C. Smith, A. Cypher and K. Schmucker, Making programming easier for children. *Interactions*, Vol. 3, No. 5, 1996, pp. 58-67.
[31] D.C. Smith, A. Cypher, and J. Spohrer, KidSim: Programming agents without a programming language, *Communications of the ACM*, Vol. 37, No. 7, 1994, pp.54-67.
[32] D.C. Smith, A. Cypher and L. Tesler, Novice Programming Comes of Age, *Communications of the ACM*, Vol. 43, No. 3, 2000, pp. 75-81.
[33] Stegecast Evaluation Version downloaded from Stegecast Home Page <http://www.stegecast.com>.
[34] J. J. G. Van Merrienboer and S. Dijkstra, The four-component instructional design model for training complex cognitive skills. In Tennyson R. D., Schott F., Seel N. M. and Dijkstra S. (eds) *Instructional Design: International Perspectives Volume 1: Theory, Research and Models*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996, pp. 427-445.