Visualisation Techniques for Learning and Teaching Programming

Lynne P. Baldwin, Jasna Kuljis

Department of Information Systems and Computing Brunel University, Uxbridge, UK

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 consider the nature, structure and function of domain-specific knowledge. Visualisation techniques may offer important insights into the learning and teaching of programming. It has been argued that conceptual models could serve to enhance learners' conceptual understanding of programming, and we describe how these may effectively be used in the teaching of programming. The methods 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.

Keywords: computer science education, learning programming, visual programming

1. Introduction

The demand for competent programmers has risen dramatically in the last several years. The 'millennium problem' has emphasised the importance of ensuring that there are IT specialists with good programming skills. The high demand for computing studies graduates in the job market both nationally and worldwide, together with the wider access to higher education generally, has prompted many students to enter into the field of computer science. Given the diverse educational backgrounds and skills of such students, together with the ever-increasing number of them in our classrooms, the role of computer science educators in teaching good programming skills is becoming ever more important.

The majority of students, even those enrolled on computer science courses, find computer programming a difficult and complex cognitive task (Guindon, 1990; Jeffries et al, 1981; Kim and Lerch, 1997; Letovsky, 1986; Simon, 1973; Mayer, 1989). The seriousness of the problem recently prompted an email debate among UK educators (`cphc-members@mailbase.ac.uk`). Academics reported their experience in using various teaching methods to get better results in teaching programming. Changing the language that is taught first does not significantly change the pass rate. Neither does using different textbooks, slowing the course down, or does alternating bottom-up versus top-down approach. Some academics opted to lower standards and/or to reduce the quantity of material taught and concentrate on 'important' topics. Because of the increase in student population the classes are large and classes of 100 – 200 students in computer programming courses are not uncommon. Usually, there is a huge vari ety in students' abilities, learning speeds, and attitudes. Attention to difficulties faced by an individual cannot be easily addressed in a large class. Therefore, educators are faced with a task of having to find an alternative way of helping students in their learning by providing a variety of modes of learning and supporting different rates of learning. One possibility is to let students teach themselves with the help of computer.

The knowledge and skills required in order to become a successful programmer are the subject of much debate, and for those engaged in the teaching of programming, it is necessary to look not only at programming knowledge learners need to develop, but to look at human

cognition, that is, how learners learn. Conclusions from this suggest that program visualisation may offer important insights into the learning and teaching of programming. The intention is to provide a system for novice users which will help them create an accurate mental model. If such a system proves to be useful it would be a good starting point from which better systems can be built.

This paper does not describe any experiments that have been carried out with regard to the learning of programming using visualisation techniques, but instead it explores the background that will inform such future research. The next section examines which programming knowledge and skills learners need to develop. The section that follows considers visual programming as a potentially more intuitive representation for programs, followed by a section which provides a brief discussion on what can be represented visually. In the conclusions the paper argues that little is known about the learning of programming and that these issues have not been given due attention in the literature to date, yet are essential in framing any empirical work.

2. Learning programming

Programming demands complex cognitive skills such as reasoning and planning. However, as Kurland *et al* (1986), note, we know little about these. Interest has centred towards trying to discover precisely which higher level thinking skills learners need in order to develop programming ability, and how those involved in the teaching of programming may best foster these in the students they teach. Thinking skills are, however, not domain-specific. Although general intellectual ability, especially logical reasoning and spatial ability, play a part in learning how to program, the difficulty in attempting to understand how such complex intellectual skills are used by learners further demonstrates the challenges that those involved in teaching such learners face in assisting them with the task. The importance of looking at the nature, structure and function of domain-specific knowledge is, as McGill and Volet (1997) state, potentially useful. They say that there is now general acceptance in the literature that learners need to acquire and effectively use the following three interrelated types of knowledge of programming: syntactic, conceptual and strategic.

Syntactic knowledge is concerned with specific facts regarding a programming language and deals with the rules governing its use, say Bayman and Meyer (1988). With this type of knowledge, learners are able to write programs that can be compiled but they do not possess the technical knowledge that allows them to design and develop programs that can effectively solve computing problems. For Linn (1985), syntactic knowledge is described as the acquisition of knowledge about language features, but both definitions concern what might be termed low-level knowledge of programming. Another kind of knowledge, conceptual knowledge, concerns the understanding of constructs and principles which govern the semantics of programming actions. By employing the syntactic knowledge of the language features, learners of programming are able to design a program, say Bayman and Mayer (1988). Linn's (1985) definition of conceptual knowledge is also related to the learning of design skills, although Linn cautions that developing a limited set of templates and procedural skills allows learners to design solutions to computer problems which are only very closely related.

The most complex knowledge that learners need to have with regard to programming is strategic. Once learners have strategic knowledge they are, say Bayman and Mayer (1988) able to solve programming problems that are more complex and beyond those that have been met before. Such strategic knowledge is necessary not only to recognise a problem but also to decompose it in order to design the phases that will subsequently be programmed, and is also essential for testing and debugging errors. Linn's (1985) definition of syntactic knowledge closely mirrors this in that it concerns the development of a wide range of problem solving skills, and that these can be used to solve programming problems that are not language specific; in short, it can be said that such learners have a robust understanding of programming. Although it is useful for those teaching programming to be able to describe programming knowledge in this way, it nevertheless points up some difficulties. Not least among these is that the learning of programming is part of learning in general, and that

in order to be able to program, complex cognitive skills such as reasoning, problem-solving and planning play their role. Assuming that those new to programming are people of reasonable intelligence, we can be fairly confident that such skills have been learned and practised earlier in life and in other areas. We do not, as yet, know how, or if, the reasoning skills involved in programming can be separated from other reasoning skills that we, as humans possess, as little is known about thinking skills in many areas of learning in a wider context.

Thinking skills are not domain-specific, and thus looking at programming needs to be seen within the broader context of learning in general. Of particular interest is to find out a great deal more about the 'readiness skills' that those beginning the task of learning to program need to have. Readiness skills are concerned with looking at learners before they begin the task; but what they intend to do with programming once they have become competent in it is also a factor in that how far learners intend to go with programming may also affect in some way how they approach the task. Little is known about how humans acquire, store and use knowledge, and this naturally includes syntactic, conceptual and strategic knowledge. Although learning programming is not easy for learners, it would seem that the task is not that much easier for those teaching it.

Those involved in the teaching of programming to those students who are new to the task, have long been criticised for failing to develop students' understanding in the key area of semantics, that is, program comprehension (De Corte et al., 1992; Haynes, 1998; Linn and Dalbey, 1985; Linn, 1985; Oliver and Malone, 1998). Shih and Alessi (1994) argue that over-emphasis on the 'how to' may not facilitate the transfer of what is learned to novel situations as it does not highlight the knowledge underlying such skills. Over-emphasis on the 'why' however, although providing learners with a wider knowledge base which can be applied in a variety of contexts, can result in a mismatch between instruction and hands-on practice. Shih and Alessi (1994) comment that theories which seek to account for how learners develop conceptual understanding offer differing explanations of the mechanisms which underlie such learning. Such differing views therefore mean that there are differing opinions about which teach ing methods may be the most appropriate (Cormier and Hagman, 1987).

Clear (1997) suggests that those involved in the teaching of programming need to reconsider their approach to teaching in light of current theories on cognition. These, he argues, may require us to adopt 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 emphasise interaction may be more powerful. Visualisation techniques, posits Clear (1997), may offer important insights into the learning and teaching of programming.

Research suggests that the human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering relationships in complex pictures than by reading text (Raeder, 1985). There has been some research into software visualisation to enhance the comprehension of algorithms and computer programs (Price at al., 1993; Stasko et al., 1997) and for debugging (Baecker et al., 1997). Carroll's (1995) and Carroll's and Rosson's (1991) work on learning to program in Smalltalk apply the concept of guided exploration. The next section discusses how visual (iconic) programming can be used to aid task of computer programming.

3. Visual programming

Visual programming uses visual expressions such as diagrams, free-hand sketches, icons or graphical manipulators (Shu, 1992). The earliest work in visual programming started with two kinds of visual programming languages: visual approaches to traditional programming languages, such as executable flowcharts, and visual approaches that departed from tradition, such as programming by demonstrating the desired actions on screen (Burnett and McIntyre, 1995). Visual systems to aid programming make use of various metaphors. Olsen et al's. (1990) visual objects consist of a set of templates including, for example, forms, modules,

statements, programs, and subprograms. Hirakawa et al.'s (1990) objects, algorithms and data structures are visualised by means of icons, data flow graphs, and spatial placement of icons, respectively. In considering the possible representations of a problem, we can adapt Bruner's (1966) classification into enactive, iconic, and symbolic representations. 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 iconic programming languages, not only is the flow of control or data represented graphically, but also icons represent the operations themselves. Iconic programming systems offer a potentially more intuitive representation for programs because they allow users to create programs and examine their execution using pictures and diagrams. They may make it easier for novices to learn how to program and, it is argued, may also be easier for people from differing language backgrounds. Such systems have to model people's understanding of programming ideas, so as to identify those qualities that make an iconic programming system easy to learn. However, programming involves many, often abstract, objects and concepts which are not easily mapped to the physical world. A key research problem is to discover new visual metaphor for representing those programming components that have no natural and obvious physical representation. Visual representation is just one aspect of a complex problem. In order to understand when to use visualisation and then to create effective visualisations we need to understand the learners' needs. Modelling learners' understandings of programming is a complex task and, according to Smith et al. (1996), the solution is to make programming more like thinking. Eberts (1994) claims that many of the most effective and accurate mental models seem to be spatial in nature.

In Cocoa (Smith et al., 1996), a children's iconic programming system, the approach to programming is to eliminate programming language syntax. Cocoa uses representations which are analogous to the objects being represented and allows these representations to be directly manipulated in the process of programming. The programmer specifies the behaviour using graphical rewrite rules. Programming by demonstration allows children to directly manipulate the representations, and graphical rewrite rules provide an understandable representation for the recorded programs. This approach would not necessarily lead to learning programming. The aim of our research is to teach would-be programmers the concepts of traditional programming. Therefore, our system should support a novice in the task of algorithmic programming where the user of the system must understand the underlying concepts of variables, operations, flow of control, subprograms, recursion, etcetera.

Through the acknowledgement of the existence of visualisation, the mental picture in the mental model, the implication is that an accurate mental model can be developed if novices use an interface incorporating graphics. Such systems have to model people's understanding of programming ideas by aiding the development of accurate mental models. The methods to enhance such development 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. The problem is how to visually represent programming concepts and how to specify algorithms in a visual language.

4. What to visualise

Programming is a complex activity requiring the acquisition of non trivial new concepts, new facts and new skills, and it involves learning relationships between many things which cannot be identified except by their relationships. Programming languages deal with the unfamiliar world of data structures and algorithms. This makes them less tractable for novices. Various diagramming techniques to represent algorithms have been used to overcome some of the problems. Usually, these representations were used to guide writing program code not necessarily done by the same person. The standardisation of graphical design notations using the flowchart was popular in some earlier

attempts (see, for example, Nassi and Shneiderman, 1973; Frei et al., 1978; Tripp, 1988). Diaz-Herrera and Flude (1980) used diagrammatic representation that visually controls flow.

Control-of-flow descriptions for programs have long been performed with flow diagrams. Like ancillary representations of computer programs, flowcharts do not necessarily provide improvement in the practice of programming. Data-flow diagrams provide an alternative kind of chart for representing programs. In data flow diagrams there are no sequencing constraints other than the ones imposed by data dependencies. They seem to be advantageous when the problem to be solved is already understood in terms of data flow. On the other hand, control-of-flow diagrams are usually preferable when emphasis is to be placed on the agents (things performing acts) of a computation rather than on the objects (data) being manipulated.

Research into learning indicates that the key to successful learning lies in organisation, representation, and structuring of knowledge (Glaser, 1990). Instead of simply replacing textual expressions with visual expressions visualisation should address these issues. Graphical representations for computing objects can be created for data, program, process, and the command object. However, when considering an appropriate visual representation, choosing suitable graphical objects and how to use them is a well-recognised research issue. Tanimoto and Glinert (1990) see the development of a good metaphor and the successful use of stylisation as two major problems of representation. The development of a good metaphor requires an analogy between a physical or mechanical family of phenomena and the phenomena important in computer programming. Stylisation is a key concept in iconic programming, because on the one hand designing icons is analogous to choosing identifier names in conventional (that is, textual) programming, while on the other hand pictures of different sizes and resolutions are required in different contexts for the same operation.

Experienced programmers differ from inexperienced students in their ways of organising domain information. There is evidence that programmers' knowledge base is structured in terms of deep functional principles of programming rather than in terms of particular language syntax (Adelson, 1981). They also form a representation of a problem in terms of smaller manageable subproblems for each they already may reuse previously developed solutions. Marco and Colina (1993) call these templates and argue that students can benefit if they are taught how to implement particular templates in different problems. Regardless of which representation is used for a problem, the use of templates is obviously very important. Reusing previously derived solutions to novel problems is basically the same type of transformation that we use when making generalisations based on instances. A predominant skill is required for program ability to generalise.

An appropriate representation can provide a basis for automating program code generation. We support the view that if the program code were automatically generated, from the representation provided by a student, the student would then gain a better understanding of the relationship between the problem and the program which solves the problem. The process of learning, aided with such a tool, would eventually lead to improved problem solving skills using computers and to better structural programming skills.

5. Conclusion

This paper has described the programming knowledge and skills that learners need to develop, and concluded that this is an area of computer science education where those involved in the teaching of programming need to further consider the nature, structure and function of domain-specific knowledge. The knowledge and skills required in order to become a successful programmer are the subject of much debate. Those engaged in the teaching of programming have not only to consider programming knowledge learners need to develop, but have also to research into how learners learn.

Formal instruction has been criticised for assisting learners in developing misconceptions and misunderstandings. Iconic programming systems offer a potentially more intuitive representation for programs. They may make it easier for novices to learn how to program, because they allow users to create programs and

examine their execution using pictures and diagrams. Such systems have to model people's understanding of programming ideas, so as to identify those qualities that make an iconic programming system easy to learn. Conclusions from this suggest that program visualisation may offer important insights into the learning and teaching of programming.

As can be seen from the above, little is known about the learning of programming, and indeed of learning in general, and, as such, this paper offers an insight into the complexities involved. These issues, we would argue, have not been given due attention in the literature to date, yet are essential in framing any empirical work. This paper does not, then, offer any 'solutions' or indeed any views as to which methods might be employed in approaching the task of finding out how visualisation does, or does not, precisely assist learners in their learning of programming. We are, at this initial stage of our research, investigating current frameworks. We are exploring and evaluating various frameworks which would be suitable for representing typical programming problems. There is no single obvious visual representation technique which would ensure learnability. Early results suggest that the chosen technique will necessarily be a hybrid of existing techniques, and augmented by new schemas. It is also anticipated that it is almost impossible to provide a generic visual representation which will enhance the development of accurate mental models through the display of all programming problems.

The empirical work that we shall later carry out will necessarily develop from this and cannot therefore be described or evaluated in a more formal way at this point in time. This is, then, an exploratory, rather than technical, paper, and seeks to focus attention on the issues with a view to opening up debate in this new and exciting field.

References

[1] B. ADELSON, Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*. **9**(4) (1981), 442–433.

[2] R. B. BAECKER, C. DIGIANO AND A. MARCUS, Software visualisation for debugging. *Communications of the ACM*. **40**(4) (1997), 44–54.

[3] P. BAYMAN AND R. E. MAYER, Using conceptual models to teach Basic computer programming. *Journal of Educational Psychology*. **80**(3) (1988), 291–298.

[4] J. BRUNER, *Theory of Instruction.* Harvard University Press, Cambridge, Mass., 1966.

[5] M. M. BURNETT AND D. W. MCINTYRE, Visual programming. *Computer.* **28**(3) (1995), 14–16.

[6] J. M. CARROLL, Making use of a design representation. *Communications of the ACM.* **37**(12) (1995), 29–35.

[7] J. M. CARROLL AND M. B. ROSSON, Deliberated evolution: Stalking the view matcher in design space. *Human-Computer Interaction.* **6** (1991), 281–318.

[8] T. CLEAR, The nature of cognition and action. *ACM SIGCSE Bulletin.* **29**(4) (1997), 25–29.

[9] S. M. CORMIER AND J. D. HAGMAN, (Eds). Transfer of Learning: Contemporary Research and Applications.: Academic Press, San Diego, CA., 1987.

[10] E. L. DE CORTE, VERSCHAFFEL AND H. SCHROOTEN, Cognitive effects of learning to program in LOGO: a one-year study with sixth graders. In *Computer-based Learning Environments and Problem Solving* (E. De Corte, M. Linn, H. Mandl and L. Verschaffel, Eds.) (1992) pp. 207–228, Springer-Verlag, Berlin.

[11] J. L. DIAZ-HERRERA AND R. C. FLUDE, Pascal/HSD: A graphical programming system. *IEEE Proceedings COMSAC* (1980), pp. 723–728.

[12] R. E. EBERTS, *User Interface Design.* Prentice Hall, Englewood Cliffs, 1994..

[13] H. P. FREI, D. L. WELLER AND R. WILLIAMS. A graphics-based programming support system. *ACM Computer Graphics*, **12**(3) (1978), 43–49.

[14] R. GLASER, Toward new models for assessment. *International Journal of Educational Research.* **14**(5) (1990), 475–483.

[15] R. GUINDON, Designing the design process: exploiting opportunistic thoughts. *Human Computer Interaction.* **5**, (1990), 305–344.

[16] C. T. HAYNES, Experience with an analytic approach to teaching programming languages. *ACM SIGCSE Bulletin.* **30**(1) (1998), 350–354.

[17] M. HIRAKAWA, M. TANAKA AND T. ICHIKAWA HI-VISUAL Iconic programming environment. In *Visual Languages and Applications* (T. Ichikawa, E. Jungert and R. R. Korfhage, Eds.) (1990), pp. 121–145. Plenum Press, New York.

[18] R. A. JEFFRIES, P. TURNER, G. POLSON AND M. E. ATWOOD, The processes involved in designing software. In *Cognitive Skills and their Acquisition* (J. R. Anderson, Ed.), (1981), pp. 255–283. Lawrence Erlbaum Associates, Hillsdale, NJ.

[19] J. KIM AND F. J. LERCH, Why is programming (sometimes) so difficult? Programming as scientific discovery in multiple problem spaces. *Information Systems Research.* **8**(1) (1997), 25–50.

[20] D. KURLAND, R. PEA, C. CLEMENT AND R. MAWBY, A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research.* **2** (4) (1986), 429–457.

[21] S. LETOVSKY, Cognitive processes in program comprehension. In *Empirical Studies of Programmers* (E. Solway and S. Iyengar, Eds.), (1986) pp. 58–79, Ablex Publishing, Norwood, NJ.

[22] M. C. LINN, The cognitive consequences of programming instruction in classrooms. *Educational Researcher* **14**(5) (1985), 14–16, 25–29.

[23] M. C. LINN AND J. DALBEY J., Cognitive consequences of programming instruction: instruction, access, ability. *Educational Psychologist* **20**(4) (1985), 191–206.

[24] R. E. MARCO AND M. M. COLINA, Programming languages and dynamic instructional tools: Addressing students's knowledge base. In *Instructional Models in Computer-Based Learning Environments* (S. Dijkstra, H. P. M. Kramer and J. J. G. van Merriënboer, Eds.) (1993), pp. 445–457. Springer-Verlag, Berlin.

[25] R. E. MAYER, The psychology of how novices learn programming. In *Studying the Novice Programmer* (E. Soloway and J. C. Spohrer, Eds.) (1989) pp. 129–159, Lawrence Erlbaum Associates, Hillsdale, NJ.

[26] T. J. MCGILL AND S. E. VOLET, A conceptual framework for analyzing students's knowledge of programming. *Journal of Research on Computing in Education.* **29**(3) (1997), 276–297.

[27] I. NASSI AND B. SHNEIDERMAN, Flowchart techniques for structured programming. *SIGPLAN Notices.* **8**(8) (1973).

[28] R. OLIVER AND J. MALONE, The influence of instruction and activity on the development of semantic programming knowledge. *Journal of Research on Computing in Education.* **25**(4) (1998), 521–533.

[29] K. A. OLSEN, B. PEDERSEN, P. HARNES AND O. J. TOSSE, A visual system to support teaching of programming. In *Visual Languages and Visual Programming* (S.-K. Chang, Ed.) (1990), pp. 277–288. Plenum Press, New York.

[30] B. A. PRICE, R. M. BAECKER AND I. S. SMALL, A principled taxonomy of software vizualization. *Journal of Visual Languages Computing.* **4**(3) (1993), 211–266.

[31] G. RAEDER, A survey of current graphical programming techniques. *IEEE Computer.* August (1985), 11–25.

[32] Y.-F. SHIH AND S. M. ALESSI, Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education.* **26**(2) (1994), 154–175.

[33] N. C. SHU, *Visual Programming.* Van Nostrand Reinhold, New York, 1992.

[34] H. A. SIMON, The structure of ill-structured problems. *Artificial Intelligence.* **4** (1973), 181–201.

[35] D. C. SMITH, A. CYPHER AND K. SCHMUCKER, Making programming easier for children. *Interactions,* September+October (1996), 58–67.

[36] J. STASKO, J. DOMINIQUE, M. BROWN AND B. PRICE (EDS.), *Software Visualisation: Programming as a Multimedia Experience.* MIT Press, Cambridge, Mass, 1997.

[37] S. L. TANIMOTO AND E. P. GLINERT, Designing iconic programming systems: representation and learnability. In *Visual Programming Environments: Applications and Issues* (E. P. Glinert, Ed.) (1990), pp. 330–336. IEEE Computer Society Press, Los Alamitos, Ca.

[38] L. L. TRIPP, A survey of graphical notations for program design: an update. *ACM SIGSOFT Software Engineering Notes.* **13**(4) (1988), 39–44.

Received: October, 2000
Accepted: November, 2000

Contact address:
Dr. Jasna Kuljis
Department of Information Systems and Computing
Brunel University
Uxbridge, Middlesex UB8 3PH
United Kingdom
Tel: +44 (0)1895 203 081
Fax: +44 (0)1895 251 686
e-mail: Jasna.Kuljis@brunel.ac.uk

LYNNE P. BALDWIN is a lecturer in the Department of Information Systems and Computing at Brunel University, UK. She gained her PhD at Brunel University, and an MA in Language and Communication at the University of East Anglia, UK. Her research interests are varied, although there is a strong emphasis on knowledge management, decision-making, and related communication issues in both industrial and educational settings. Her email address is <Lynne.Baldwin@brunel.ac.uk>.

JASNA KULJIS is a Senior Lecturer in Computer Science at the Goldsmiths College of University of London. She has a B. Sc. in Mathematics from Zagreb University, an M. S. in Information Science from Pittsburgh University and a Ph. D. in Information Systems from the London School of Economics. She is currently researching into human-computer interaction and in visual programming. She has published widely in many aspects of computing. Her email and web addresses are <jasna.kuljis@brunel.ac.uk> and <www.brunel.ac.uk/~csstjjk>.