

# Using Software to Teach Computer Programming: Past, Present and Future

by  
**Paul Mulholland**  
**Marc Eisenstadt**  
**Knowledge Media Institute**  
**The Open University**



Over the past twenty years, we have undergone a change of perspective in the way we teach programming. Having begun by worrying in detail about the needs of novices, and trying to understand their problems and misconceptions, we developed a range of environments and automated debugging assistants to help them (this work is described in detail in (Eisenstadt, *et al.*, 1992). Our empirical studies (Eisenstadt, *et al.*, 1984; Eisenstadt & Breuker, 1992) suggested that algorithm design and planning were not nearly as big an obstacle to our students as the lack of a clear execution model. For example, in tackling the averaging problem used by PROUST (Johnson & Soloway, 1985), our subjects had no trouble computing averages, but their algorithms were based on an intuitive approach of applying nested functions to aggregate data objects (i.e. first add up all the items, then tally them separately). The stumbling block was the mismatch between the subjects' intuitive approach and the rather artificial constraints imposed by the behaviour of a Pascal WHILE loop, and a critical issue for us was how to impart a clear model of this constrained behaviour. We therefore decided to move our emphasis to the “software maintenance” side of

## Introduction

programming, as opposed to the “design” and “planning” sides. We realised eventually that the debugging/maintenance needs of experts were fundamentally the same as the pedagogical needs of our novices: both needed to see in a perspicuous fashion what was happening during program execution, though at different levels of granularity. This resulted in a shift in our emphasis from automated debugging assistants to software visualization. To justify this shift, let’s consider the evolution of our view in detail.

### Learning by debugging

In 1976, we faced the challenge of teaching programming in very adverse circumstances: we wanted to teach AI programming to Psychology students at the UK’s Open University. Our students were (a) computer-illiterate or computer-phobic, (b) working at home with no computer hardware, and therefore having to attend a local study centre to use a dial-up teletype link to a DEC system-20, (c) studying Psychology with no intention of learning programming, (d) only allocated a period of two weeks to get through the computing component embedded within a larger Cognitive Psychology course. Our approach (described in Eisenstadt, 1983) was to design a programming language called SOLO (essentially a semantic-network variant of LOGO) which enabled students to do powerful things on the first day, embed this language in a software environment which corrected “obvious” errors (such as silly spelling mistakes) automatically, make the workings of the underlying virtual machine both highly explicit and very visible, and develop a curriculum sequence which from start to finish tried to motivate the student by highlighting the relevance of each programming task to the student’s main academic interest—cognitive psychology. Visibility of the underlying virtual machine was achieved by printing out changes to the semantic network as they were made by the user, although the innards of control flow were not particularly visible in the sense that we describe below.

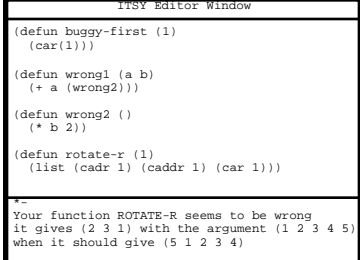
Our students wrote plenty of buggy SOLO programs, and we were highly motivated to understand the nature of our students’ problems, and also to develop automatic bug detectors and intelligent tutoring systems which capitalised on this understanding. We studied and modelled our student’s misconceptions in some detail (Kahney & Eisenstadt, 1982; Kahney, 1983; Kahney, 1992), finding that deep-seated problems such as understanding recursion were often due to a student’s failure to map analogically from the detailed curriculum examples to the programming task at hand. Indeed, Conway and Kahney (1987) found that

explicit instruction about how to perform the analogical mapping was of direct benefit to the students. We extended our studies to other languages including Pascal (Eisenstadt *et al.*, 1984; Eisenstadt & Breuker, 1992), and found that although the fundamentals of iteration were not problematic to novices, the contorted mapping to specific language constructs was problematic: novices seemed to prefer to apply nested functions to aggregate data objects rather than cycling through individually-indexed objects. All of these studies confirmed our view that novices employed quite sensible models of the world, but that programming language instructors in general (including ourselves) consistently failed in helping novices to map their pre-existing models onto the specific ones required to deal with programming. Some of our studies (e.g. Kahney & Eisenstadt, 1982) directly led to a revision of our SOLO curriculum.

We built a variety of automatic debuggers for SOLO (Laubsch & Eisenstadt, 1981; Hasemer, 1984; Hasemer, 1992; Laubsch & Eisenstadt, 1992) and other languages, including Lisp (Domingue, 1987) (see figure 1) and Pascal (Lutz, 1992). The approaches varied, but all involved some sort of cliché detection and near-miss analysis, and were strongly influenced by the MIT Programmer's Apprentice project (Rich, *et al.*, 1979; Rich & Waters, 1988; 1990). As AI researchers, we were very pleased when our programs could automatically detect a bug and make sense of the student's problem. As programming language instructors, however, we faced an awkward dilemma. Explaining the root cause of the bug to the student seemed inordinately difficult, because it involved concepts that the student didn't really understand (if the student had understood, he or she wouldn't have been caught by that particular bug in the first place). This is a standard pedagogical problem, namely how to help the student leap across "islands of understanding", and was regrettably outside the scope of our work on automatic debugging. When we tried out our systems on our residential summer school students, we observed that what appeared to help them the most was *showing* them (on a blackboard) what their program was doing and why. Indeed, they didn't particularly need the automatic bug detectors once they could see what was really happening.

At the same time, our students were feeling frustrated about reaching the limit of SOLO's powers very quickly, and not being able to extend their programs to handle more complex tasks. Rather than continue with SOLO development, we decided to teach Prolog, which exhibited many of the same pattern-matching

### The ITS years



```

ITSY Editor Window
(defun buggy-first (l)
  (car(l)))

(defun wrong1 (a b)
  (+ a (wrong2)))

(defun wrong2 ()
  (* b 2))

(defun rotate-r (l)
  (list (cadr l) (caddr l) (car l)))
--
Your function ROTATE-R seems to be wrong
it gives (2 3 1) with the argument (1 2 3 4 5)
when it should give (5 1 2 3 4)

```

Figure 1. ITSY adapted from Domingue (1987).

capabilities of relevance to our students, but was much more powerful, and quite widely used in the AI community. We were working on a project at the time to develop a graphical debugger for professional Prolog users, and decided it would be useful to “dovetail” our teaching and research activities. This meant trying to develop a cradle-to-grave environment that would be useful for our novices, but which extended all the way to the capabilities of professional Prolog programmers. “Seeing what was happening” was a very useful way to overcome some (though not all) of the bottlenecks we had witnessed among the novices who were having trouble mapping their real world models onto a programming framework. At the same time, our experts needed to “see the wood for the trees,” i.e. to have a way of homing in quickly on trouble spots. The difference between the viewpoints required by the two audiences was not simply one of physical scale size, such as might be provided for free on a graphics workstation, but rather one of *abstraction level*, which required careful attention. The end result of this line of work was TPM, The Transparent Prolog Machine (Eisenstadt & Brayshaw, 1988; Brayshaw & Eisenstadt, 1991; Kwakkel, 1991; chapter X) and its accompanying textbook and video curriculum material (Eisenstadt, 1988; Eisenstadt & Brayshaw, 1990).

### The SV story

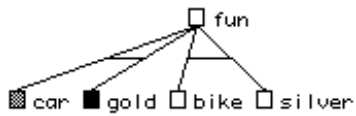


Figure 2. Example of a TPM coarse-grained view of program execution.

Our work with TPM (see figure 2) suggested that it might be possible to cater for quite a wide audience and a wide range of problems by focusing on the common thread which ran through the entire learning experience: visualization. We were developing video-based teaching material for novice Prolog programmers at the same time as we were implementing graphics facilities for helping experts observe a 2- or 3-thousand node search space. Only by forcing these two paths to converge could we cater for the “upwardly mobile student” who learned about Prolog in the early phases and then went on to become a serious Prolog user. By stepping back to think about the end-point of the learning trajectory, we were able to provide a useful spin-off for those at the beginning of the trajectory.

At the same time as our work on program visualization was developing, some of our colleagues at the Open University had come up with an approach to constructing Intelligent Tutoring Systems that they called Guided Discovery Tutoring (Elsom-Cook, 1990). This approach excited us, because it combined ITS work with a strong emphasis on software environments in which students were free to explore and develop in their own way, but under the benign watchful

eye of a tutorial assistant. This was in contrast to the earlier ITS work on programming which embodied a different “instruction-based” paradigm. In that earlier paradigm, students were led through a structured curriculum, embarking on well-defined tasks, and even designing and implementing their programs according to a model of “good practice” which had been painstakingly captured by the ITS designers (indeed this very model was critical because it facilitated automated diagnosis when students go wrong). The “instruction-based” paradigm appears to embody clean, top-down, robust and reliable software-engineering practice (i.e. it aims to get the design and specification right from the beginning), but in reality it can act as a straight-jacket for many students. Although it is hard to argue against such a noble-sounding path, we stress that it is not the *only* path, and moreover it is simply not appropriate for all students. Nor is it the path followed by most professional programmers. (This could be why we have a “software crisis” at the moment, but we doubt it. There are many other issues involved, which are beyond the scope of this chapter to address.)

We said at the beginning of this section that our overall emphasis shifted from the “design” and “planning” aspect of programming towards the “software maintenance” side. This would appear to be encouraging students to engage in brute force “hacking” (in the old sense) rather than in good software engineering practice. However, the tenets of Guided Discovery Tutoring, like the seminal work of Papert (1980), suggested to us that we could indeed trust students to do some unstructured exploration, and gently nudge them in a different direction if they were ready for it. This “gentle nudge” could take the form of a whole curriculum sequence, or it could be a matter of just showing the students a particular way of thinking about program execution. To explore this assumption we carried out a number of empirical studies aimed at ascertaining the extent to which students were able to benefit from a range of SV environments. The next two sections will outline the empirical findings and their ramifications for the educational role of SV.

Our recent work provides support for the argument that no SV will ever be universally superior across all kinds of users and tasks (Mulholland, 1995; chapter X). Two important factors influencing performance are (a) the programmers' expertise with the programming language in question, and (b) their level of familiarity with the SV itself. It does seem possible, however, that a particular SV could be most suitable for a novice population. This is for two

**The truth about SV**

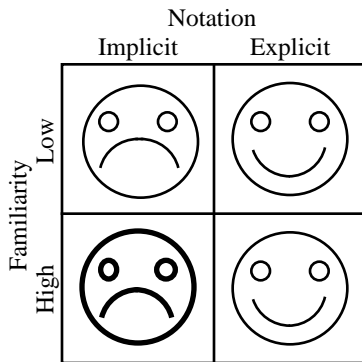


Figure 3. The effect of notational familiarity on novices using explicit and implicit information.

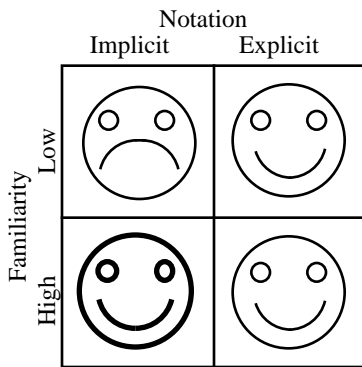


Figure 4. The effect of notational familiarity on experts using explicit and implicit information.

reasons. Firstly, a crucial factor in how well a novice is able to use an SV is the extent to which the notation helps to combat misunderstandings. A notation successful in this venture will undoubtedly prove to be a reasonable tool. Another important factor is how well the SV facilitates the utilisation of strategies the novice is able to employ. Strategies relying on explicit information such as mapping between the trace and the code, and control flow and data flow strategies are of particular importance.

It seems far less likely that an SV will be most appropriate for all experts irrespective of the kinds of task they wish to perform. This is for two reasons. Firstly, experts have a sound knowledge of the execution model and therefore do not suffer from the associated misunderstandings. As a result, the ability of the SV to present information in a way that deals with potential misunderstandings is far less important. Secondly, experts are able to develop a much more elaborate collection of strategies, many utilising implicit information. With experience of an SV the experts are able to tune their strategies to fit the features of the SV. For experts the relative efficacy of SVs will to some extent always be a matter of taste.

The important distinction between novices and experts suggested by the results is how increased familiarity with the SV may affect the nature of the strategies used. Once novices have become reasonably familiar with the notation it seems that any further exposure to the SV without a corresponding shift in expertise will not alter the way in which the SV is utilised. Their use of the SV is restricted by the kinds of strategies they are able to employ. Their level of expertise confines them to using strategies which rely on the information explicit within the trace. The effect of notational familiarity on novices using explicit and implicit information is summarised in figure 3. Novices also seemed not only affected but confined by the kinds of misunderstandings they have of the language. Protocol evidence revealed many instances where when confronted with a disparity between the SV and their incorrect assumptions the subjects would reinterpret the display in such a way as to be consistent with their expectations rather than the disparity leading them to challenge their expectations and the misunderstanding from which they originated.

Familiarity does, however, affect the way in which experts make use of the SV. Initially the expert will rely on strategies drawing on explicit information, as

they fully develop an understanding of how the notation of the SV corresponds to the execution model. With greater experience the expert is able to draw on implicit information within the trace as Gestalts or visual clichés become noticeable. The effect of notational familiarity on experts using explicit and implicit information is summarised in figure 4. When using an SV to provide an explanation, experts experienced with the SV are able to draw attention to function related information at an earlier stage as they are already aware how the higher level abstractions of the program map onto the patterns found within the display. Experts unfamiliar with the SV are required to initially work with explicit information and tie the declarative account to the SV at a later stage.

These results allow us to comment on whether text or graphics is the most appropriate medium for displaying such information. As we have seen, experts given sufficient familiarity are able to incorporate implicit information into their strategies. Novices, it seems, are not. This suggests that a notation which necessitates access to implicit information is not suitable for programmers of low expertise or those intending to use the technology for a short period of time. As a graphical notation will probably rely more heavily on implicit information, evidence suggests that these should be reserved until the student has acquired a level of expertise which permits them to appreciate abstractions. Exposing the novice to such an SV would be to encourage them to perform like mini-experts rather than learning through the application of strategies appropriate to their skill level, though whether this is necessarily worse would require further study.

These results are consistent with the literature on novice expert differences and the development of expertise. For example, Chi *et al.* (1981) found that novices tend to focus on the surface features of the language whereas experts tend to use higher level structural representations. Similarly, Davies (1994a) found that experts tend to externalise higher level structures during program comprehension but novices tend to externalise code-level information. There is a wealth of evidence from a number of domains to suggest that these performance differences are due to the knowledge structures of experts and novices differing qualitatively rather than quantitatively and that knowledge has to be restructured during an intermediate stage before progressing to full expertise. For example, Lesgold *et al.* (1988) found that during some phase in the development of expertise in radiological diagnosis the number of errors made temporarily increases. They suggested this kink in the learning curve could be a consequence of a

### Finding the role of SV within education

restructuring of the knowledge underlying performance. Similarly, Davies (1994b) in his investigation of recall for focal and nonfocal lines in Pascal programs found expertise to develop in a non-linear fashion

Acceptance that there is no software panacea suggests new ways in which the proper role for SV within an educational setting could be established. This covers two related questions:

#### 1) How should SV be incorporated into computer programming education?

As considered above, an important design aim in the development of SVs has been to find a single representation of a programming language suitable for all levels of expertise. Not only do the presented findings suggest none of the SVs studied can confidently claim universal superiority but that because of the qualitative differences between programmers of different levels of expertise such an aspiration will inevitably fail.

This could be empirically studied by comparing two approaches to incorporating SV into programming education. One approach could use a single SV throughout the course. We have used (though not evaluated) this approach in the development of a Prolog course which uses the TPM notation throughout all materials (Eisenstadt & Brayshaw, 1987). The work presented above suggests that TPM may not be a very suitable SV for the early novice, though it could be argued that the difficulties confronting the early novice are possibly outweighed by the extra cost of having to transfer between different notations during the course, should TPM be presented at a later stage. An alternative view would be that novices will learn more effectively through a notation suited to their characteristics and strategic capabilities than through a notation which requires them to aspire to expert strategies from an early stage. For example, Prolog could be taught by initially using an SV such as Theseus (Mulholland, 1995) (see figure 5) and transferring to TPM when the students are reaching the stage of being able to externalise their knowledge and use abstractions. It could be argued that the change in notation will not adversely effect the students as their knowledge is undergoing restructuring which will parallel the notational transition.

#### 2) Could SV be used specifically to support the transition between novice and expert?

```

• ? fun(What)
  «1 fun({What=X})
    ? car(What)
      +1 car({What=mini})
        ? gold(mini)
          -d gold(mini)
            ^^^^^^^^^^^^^^^^^
• >2 fun({What=X})
• ? bike(What)
• +1 bike({What=honda})
• ? silver(honda)
• +1 silver(honda)
• +2 fun({What=honda})

```

Figure 5. Example of a Theseus trace.



The dual SV approach to programming education discussed above would aim to mirror the transition from novice to expert in the learning materials. A more specific issue is how SV could support the knowledge restructuring stage as students move from novice to expert. Karmiloff-Smith (1979/94) provides some evidence that representational diversity is a crucial component in the attainment of expertise. Her theory of human development has highlighted the importance of what she termed Representational Redescription (RR). RR is the process of learning by iteratively rerepresenting knowledge in different formats. Karmiloff-Smith's observations of children have found that the child will initially develop competence at some skill. The focus of this stage is what she terms "behavioural mastery" rather than the development of a deeper level of understanding. The child will then "unpack" the structures underlying that competence in order to make them more explicit to the individual. This occurs by initially making the new internal (rather than the external) representation the main driving force. During this phase new kinds of errors previously absent appear as a result of the incompleteness of the internal representation. Later the external and internal representations become reconciled leading to a new competence underpinned by a deeper understanding. This process of unpacking knowledge to refine the internal representation is facilitated by iteratively redescribing the knowledge in different internal representational formats.

A similar process may also apply to the development of computer programming expertise. We found that novices will often use strategies such as mapping between the SV and the code without having a high level understanding of the constructs being compared (Mulholland, 1995). This process sometimes led to errors but also often led to the subject making the correct response for incomplete reasons. For these subjects, their next goal would not be an increase in competency, but rather an examination of the strategies being used and an understanding of the limits of their applicability.

Unpacking competent behaviour to foster a deeper understanding is a process SV may be able to support. SVs can be used to provide abstract and diverse representations of the same execution. For example, a diverse range of representations of Parlog execution are found in the Multiple Representation Environment (Brayshaw, 1994). These differ both in terms of the notational formalisms used and the levels of abstraction on which they work. Similarly, Algorithm Animation environments such as Balsa (chapter X), TANGO

(Stasko, 1990) and VIZ (Domingue *et al.*, 1991) provide a range of diverse perspectives on program execution. These could provide external support and prompting for the necessary internal redescription processes. The diverse range of abstractions available could be used to suggest ways in which the students could redescribe their knowledge. A similar role could be found for bug location agents which can be used to delegate certain kinds of search and inference processes to the software environment (Brayshaw, 1993). This could facilitate the externalisation and examination of internal representations and processes.

## Conclusion

An important consequence of this work would be to identify the level and kind of support necessary for each stage of the learning process. Many references have been made to the role free-exploratory learning could play in computer programming education, particularly when supported by a rich environment (e.g. Ramhadan, 1992). The kind of research outlined above would aim to identify the stages during the learning process where exploration was most appropriate. This would most likely parallel the stages of expertise where the students are able to externalise and evaluate their own knowledge in light of what is being shown to them, rather than when the students merely reinterpret incoming information in order to preserve some consistency with their own expectations.

Such an issue will become more important as new technology such as the internet becomes applied to distance education, where multimedia course materials can be used to fuller effect and the opportunity exists for the student to take a more independent role within the learning process. However, the introduction of rich interactive environments will not guarantee a fruitful learning experience. Rather, the nature of the environment and the level of free exploration must be appropriate to the characteristics of the learner.

## References

- Brayshaw, M. (1993). Intelligent Inference for debugging concurrent programs. In V. Marik (Ed.) *Conference on Expert Systems and Database Applications (DEXA-93)*, Berlin: Springer-Verlag.
- Brayshaw, M. (1994). Information Management and Visualization for Debugging Logic Programs. PhD Thesis, Human Cognition Research Laboratory, The Open University, Walton Hall, Milton Keynes, UK.
- Brayshaw, M. and Eisenstadt, M. (1991). A Practical Tracer for Prolog. *International Journal of Man-Machine Studies*, 42:597-631.

Chi, M. T. H., Feltovich, P.J. and Glaser, R. (1981). Categorisation and representation of physics problems by experts and novices. *Cognitive Science*, **5**, 121-152.

Conway, M. A. and Kahney, H. (1987). Transfer of Learning in Inference Problems: Learning to Program Recursive Functions. In J. Hallam and C. Mellish (Eds.), *Advances in Artificial Intelligence*.

Davies, S. P. (1994a). Externalising information during coding activities: Effects of expertise, environment and task. In C. R. Cook, J. C. Scholtz, J. C. Spohrer, (Eds.) *Empirical Studies of Programmers: fifth workshop*. Norwood, NJ: Ablex.

Davies, S. P. (1994b). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human Computer Studies*, **40**, 703-726.

Domingue, J. (1987). ITSY: An Automated Programming Advisor. HCRL Tech Report No. 22, Open University.

Domingue, J., Price, B. and Eisenstadt, M. (1991). Viz: A Framework for Describing and Implementing Software Visualization Systems. In Proceedings of *NATO Advanced Research Workshop: User-centred requirements for Software Engineering Environments*, Sept 1991.

Eisenstadt, M. (1983). A User Friendly Software Environment for the Novice Programmer. *Communications of the ACM*, **26** (12).

Eisenstadt, M. (1988). *Intensive Prolog*. Associate Student Central Office (Course PD622), The Open University, Milton Keynes, UK: Open University Press.

Eisenstadt, M. and Brayshaw, M. (1987). An integrated textbook, video and software environment for novice and expert Prolog programmers. In E. Soloway and J. Spohrer (Eds.), *Understanding the novice programmer*. Hillsdale, NJ: Erlbaum.

Eisenstadt, M. and Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, **5** (4), 277-342.

- Eisenstadt, M. and Brayshaw, M. (1990). A fine grained account of Prolog execution for teaching and debugging. *Instructional Science*, **19** (4/5), 407-436.
- Eisenstadt, M. and Breuker, J. (1992). Naive iteration: an account of the conceptualizations underlying buggy looping programs. In M. Eisenstadt, M. T. Keane and T. Rajan (Eds.), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.
- Eisenstadt, M., Breuker, J. and Evertsz, R. (1984). A cognitive account of "natural" looping constructs. In *Proceedings of the First IFIP Conference on Human-Computer Interaction, INTERACT '84*, London, 173-177.
- Eisenstadt, M., Price, B. A., and Domingue, J. (1992a). Software Visualization: Redressing ITS Fallacies. In *Proceedings of NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming*, Genova, Italy.
- Elsom-Cook, M. (1990). (Ed.) Guided discovery tutoring : a framework for ICAI research. London : Chapman.
- Hasemer, T. (1984). MACSOLO/AURAC: A programming environment for novices. In *Proceedings of the 6th European Conference on Artificial Intelligence*. Pisa, Italy.
- Hasemer, T. (1992). Syntactic debugging of procedural programs. In M. Eisenstadt, M. T. Keane and T. Rajan (Eds.), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.
- Johnson W. L. and Soloway, E. (1985). PROUST: An automatic debugger for Pascal programs. *Byte*, **10** (4), 170-190.
- Kahney, J. H. (1983). Problem solving by novice programmers. In T. R. G. Green, S. J. Payne and G. C. van der Veer (Eds.), *The psychology of computer use*. London: Academic Press.
- Kahney, H. (1992). Some pitfalls in learning about recursion. In M. Eisenstadt, M. T. Keane and T. Rajan (Eds.), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.

Kahney, J. H. and Eisenstadt, M. (1982). Programmers' mental models of their programming tasks: The interaction of real-world knowledge and programming knowledge. In *Proceedings of the Fourth Annual Cognitive Science Society Conference*, Ann Arbor, Michigan, 143-145.

Karmiloff-Smith, A. (1979). Micro- and macrodevelopmental changes in language acquisition and other representational systems. *Cognitive Science*, **3**, 91-118.

Karmiloff-Smith, A. (1994). Precis of Beyond Modularity: A developmental perspective on cognitive science. *Behavioural and Brain Sciences*, *17*, 4, 693-745.

Kwakkel, F. (1991). TPM for Macintosh. Human Cognition Research Laboratory, Open University, Walton Hall, Milton Keynes, UK.

Laubusch, J. and Eisenstadt, M. (1981). Domain specific debugging aids for novice programmers. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, San Mateo, CA: Morgan Kaufmann, 964-969.

Laubusch, J. and Eisenstadt, M. (1992). The automatic debugging of recursive side-effecting programs. In M. Eisenstadt, M. T. Keane and T. Rajan (Eds.), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.

Lesgold, A., Rubinson, H., Feltovich, P., Glaser, R., Klopfer, D. and Wang, Y. (1988). Expertise in a complex skill: Diagnosing X-ray pictures. In M. T. H. Chi, R. Glaser and M. J. Farr (Eds.) *The nature of expertise*. Hillsdale, NJ: Erlbaum.

Lutz, R. (1992). Plan diagrams as the basis for understanding and debugging Pascal programs. In M. Eisenstadt, M. T. Keane and T. Rajan (Eds.), *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence*. Hove, UK: LEA.

Mulholland, P. (1995). *A framework for describing and evaluating Software Visualization Systems: A case-study in Prolog*. PhD Thesis, Open University.

Papert, S. (1980). *Mindstorms: Children, computers and powerful ideas*. Brighton, UK: Harvester.

Ramadhan, H. A. (1992) *Intelligent systems for discovery programming*. CSRP 254, Department of Cognitive and Computing Science, University of Sussex.

Rich, C., Schrobe, H. E. and Waters, R. C. (1979). Overview of the Programmer's Apprentice, In Proceedings of the Sixth International Joint Conference on Artificial Intelligence, pp 827-8.

Rich, C. and Waters, R. C. (1988). The Programmer's Apprentice: a research overview. *Computer*, **21**, 11.

Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice*. London: Addison-Wesley.

Stasko, J. T. (1990). Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, 27-39.