Chapter 3.2

Expert Programming Knowledge: A Strategic Approach

David J. Gilmore

Psychology Department, University of Nottingham, Nottingham NG7 2RD, UK

Abstract

This chapter considers an alternative to the 'programming plan' view of programming expertise, namely that expert programmers have a much wider repertoire of strategies available to them as they program than do novices. This analysis does not dispute that experts have 'plan-like' knowledge, but questions the importance of the knowledge itself versus an understanding of when or how to use that knowledge. This chapter will firstly examine evidence which challenges the completeness of the plan-based theory and then it will look at evidence which reveals explicit strategic differences. As the studies are presented a list of strategies available to experts will be maintained. Although the emphasis of this section of the book is on expert performance, the chapter concludes with a brief look at studies of novices, since the strategic approach includes the assumption that the strategies used by novices are different from those available to experts. From all these studies it will be seen that experts choose strategies in response to factors such as unfamiliar situations, differing task characteristics and different language requirements, whilst many problems for

Psychology of Programming
ISBN 0-12-350772-3

Copyright © 1990 by Academic Press Limited
All rights of reproduction in any form reserved

novice programmers stem not only from lack of knowledge, but also from the lack of an adequate strategy for coping with the programming problem.
The previous chapter has presented studies of expertise in computer programming that have concentrated on the content and structure of expert knowledge. The dominant concept has been the 'programming plan', which has been proposed as the experts' mental representation of programming knowledge, and which has been used in the development of programming tutors and environments for novice programmers.

This chapter examines those aspects of expertise that are not easily explained by plan-based theories. It is intended that the strategic approach described in this chapter should be seen as a complementary, rather than alternative, explanation of expertise. As the various studies are described a list of important strategies which programmers may use will be developed. The studies that are to be presented reveal problems with two implications of the plan-based approach:

(1) that 'programming plans' provide a complete explanation of expert programming behaviour;

(2) that a novice can be defined as someone who does not possess this expert knowledge.

An important, but often implicit assumption of the plan-based theorists is that the cognitive processes underlying programming are relatively straightforward. Often based on ideas from artificial intelligence, these processes are taken to be general problem-solving skills (cf. SOAR, Laird *et al.*, 1987; ACT* , Anderson, 1983), which a novice programmer also possesses. The learning of computer programming is the acquisition of the appropriate knowledge structures for the problem-solving skills to use. These knowledge structures may then be labelled 'plans' (see Chapter 3.1 for more details), though other possibilities exist. The critical feature is that expertise is seen as the acquisition of knowledge.

The alternative position is that expertise in programming may involve a variety of cognitive processes which, coupled with changes in knowledge, can give rise to a choice of different methods for solving any particular programming problem. These different methods can be termed strategies. The critical feature of the strategy argument is that observations of novice-expert differences can be caused by either knowledge differences, processing differences, or both.

Any assessment of this argument has two important strands. Firstly some evidence questioning the plan-based theory will be presented (evidence for the theory has been covered by the previous chapter) and then evidence of strategic differences will be described. For these reasons it is important that the evidence for alternative aspects to expertise beyond the plan-based theories is carefully considered

1 Generalizations of 'programming plan' theories beyond Pascal

The majority of research on programming plans has examined Pascal experts, since they are easily found in universities. However, if the theory is a truly accurate characterization of programming expertise the concept should generalize to other languages and paradigms.

```
program prob12;
vars depth, days, rainfall:integer;
     average:real;

begin
for days := 1 to 40 do
begin
depth := 0
writeln("Noah, please enter todays rainfall:");
readln(rainfall);
rainfall := rainfall + depth;
end;
average := depth / 40;
writeln("Average is", average);
end.
```

Figure 1: An example Pascal program with plans cued using different fonts (from Gilmore and Green, 1989). This program contains two errors.

One experiment that looked at the generalization issue was that by Gilmore and Green (1989), who examined the nature of programming plans in Pascal and Basic. They examined the ability of final-year undergraduate programmers (not expert, but comparable with groups used in similar research) to detect different types of errors, under different conditions of program presentation. The relevant condition for the current discussion was one in which the plan structure of the program was highlighted using different coloured highlighting pens. Figure 1 provides an example of this condition, using different fonts rather than different colours. The important result was that, for the Pascal programmers only, the highlighting led to a greater detection of plan-related errors (27% improvement) and to no improvement in the detection of other types of error (1% decrease). Thus, the Pascal programmers were responsive to plan structures, when performing plan-related tasks.

This result confirms previous work on the programming plans, though the task-specific nature of the effect is a departure from previous research. However, the results for the Basic programmers were quite different. In general they were not as responsive to structural cues as were the Pascal programmers, and they were least responsive to the plan-structure cues, which led to a 6% decrease in detection of plan-related errors and a 3% decrease in the detection of other errors.

Also important, given arguments that plans are the major aspect of expertise, was the result that Pascal programmers were also responsive to control-structure cues, but only when detecting control-flow errors (37% improvement, cf. 4% improvement on other errors). This indicates that according to the information available from the program the programmers were able to switch between different perspectives on the code.

From this study it became clear that the literal content of programming plans

is not transferable between languages, even within the procedural paradigm. This means that we must either acquire a more precise theory of the development of programming plans, so that we can direct the differences across languages, or else we must perform new analyses of plan structures for every programming language. Gilmore and Green explained the effect in terms of difference in the *role-expressiveness* of Pascal and Basic.

Davies (1990) was unconvinced by this explanation of the language differences and repeated the experiments using a single language. He used the same methodology, but compared Basic programmers with and without formal instruction in software design skills. His results show that although programmers who had experience of structured programming techniques were responsive to plans (25% improvement in error detection), those programmers without experience of structured programming showed no such improvement (3%).

Since we have no information about the design skills of Gilmore and Green's subjects we cannot tell whether their experiment demonstrated the same effect as Davies and that what they interpreted as language differences were in fact due to differences in exposure to structured programming. Nevertheless, these results together pose a number of questions for advocates of plans as a theory of expertise in programming. These include 'where do plans come from?' and 'how are plans used in program generation?'.

The latter has been examined by Bellamy and Gilmore (1990) who collected protocols from a number of experts in different languages whilst generating a few simple programs. They attempted to compare Rist's model of plan use during coding (Rist, 1986) with the *'Parsing-Gnisrap'* model of Green *et al.* (1987). Unfortunately the results did not particularly support either model, since the evidence for plan use was poor. It seems that we are not yet at a stage where theories of the use of plans can be adequately described, since there is no usable, objective definition of a programming plan.

These three studies challenge the simplistic view of plans, but they do not give any cause to reject the concept completely. They provide a basis for arguing that much more research is needed on the psychological basis for plans, before we proceed with applying the theory. Furthermore, there is an increasing amount of research that reveals aspects of expertise which cannot be accounted for by theories of programming plans. It is to this literature that we now turn.

2 Alternative perspectives on expertise

The main problem with knowledge-based theories of expertise is that the learning process acquires knowledge about programming, not knowledge about how to do programming. Kolodner (1983) neatly captures this difficulty:

> even if a novice and an expert had the same semantic knowledge ...,
> the expert's experience would have allowed him to build up better episodic definitions
> of how to use it.

In fact, one of the early papers on programming plan theories of expertise (Chapter 3.1) included the learning of 'programming discourse rules', rules about conventions and styles in programming. Perhaps unfortunately, the emphasis since that paper has been almost solely on plans, leaving others to investigate episodic knowledge about how to do programming.

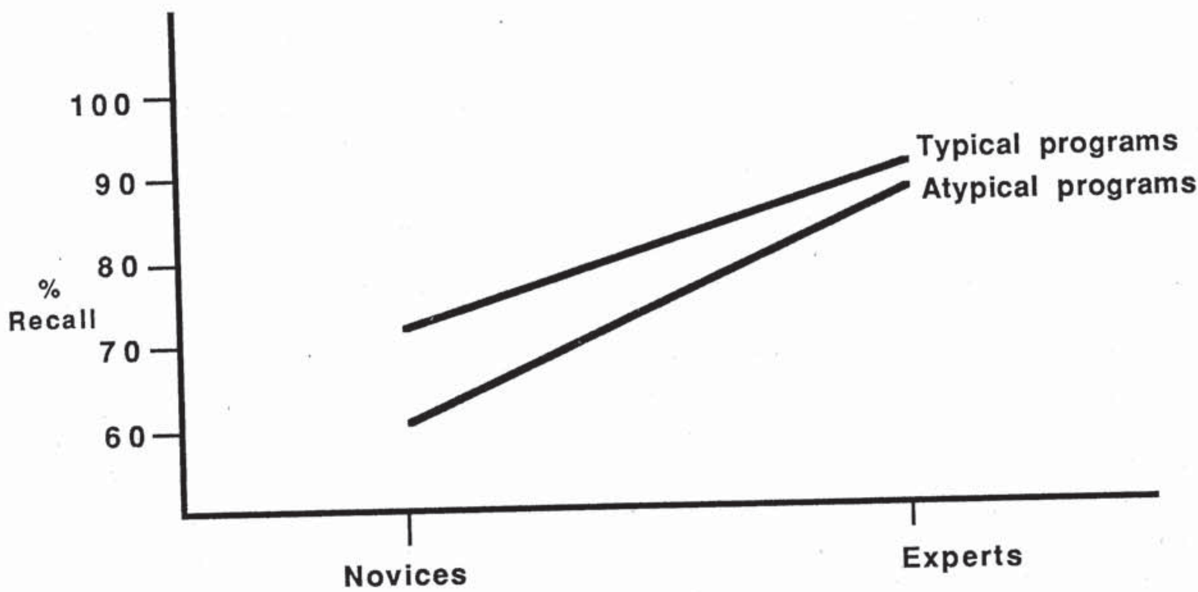

Figure 2: Widowski's (1987) results, showing that novices are, if anything, more affected by typicality than are experts.

3 Comprehension processes

The traditional application of the Chase and Simon (1973) chess research (see Chapter 1.4) is to compare novices and experts on well-ordered versus randomly ordered programs. But, since the effect is understood to be based on the expert's ability to readily extract meaningful chunks, the same paradigm can be applied to well-ordered programs that are more or less meaningful (typical or atypical). The knowledge-based theory would predict that novice-expert differences should be diminished with the atypical programs since neither the novices nor the experts will possess the plans to process it.

Widowski (1987) performed this comparison, varying both the degree of typicality (assessed through semantic complexity) and *structural complexity* (using the McCabe (1976) measure of syntactic complexity). He compared thirteen Pascal novices with thirteen Pascal experts in a programming environment that allowed them to view only one line of the program at a time. This enabled the collection of data about both the process and the results of comprehension. The subjects' task was to comprehend the program and then to reconstruct it from memory.

The recall results showed significant effects of expertise and of structural and semantic complexity. There was an interaction between the two forms of complexity, with the impairment due to semantic complexity being much greater in the structurally complex programs. But, the most striking result was the complete absence of an interaction between expertise and complexity. Contrary to the hypothesis, experts were better than novices for both typical and atypical programs. In fact, if anything the difference was greater for the atypical programs (see Figure 2).

On the data from the comprehension process there was evidence of plans guiding the experts' comprehension on the stereotypical programs, with a significant interaction between expertise and semantic complexity. On the atypical programs, experts seemed able to shift to a quite different strategy of comprehension. For novices

the same strategy of comprehension seemed to be used for all programs. Structural complexity had no effect on comprehension processes.

In a second study, a similar task was used, but with verbal protocols collected during the comprehension process as well. In this the results of the first study were replicated. In the comprehension process two strategies were identified: control-structure oriented and variable-oriented. The main results were thát experts varied the amount of structure-oriented processing according to the complexity of the program, but that they consistently used the variable-oriented strategy more than novices.

The identification of these two strategies, and the ability of experts to use either as needed, is consistent with the task-specific results of Gilmore and Green described above. Although Gilmore and Green did not explicitly address the comprehension process, it is possible to take all these results together, which suggests that the focus of a programmer's comprehension strategy is affected by the program presentation format (Gilmore and Green, 1989), the program's complexity (Widowski, 1987) and by the programmer's knowledge (Davies, 1990).

Whereas Widowski focused on the *process* of comprehension, Pennington (1987a) has looked at the mental representations that programmers form of a program. In her study she used a range of types of comprehension questions, each type accessing a different type of information from the program (e.g. control flow, data flow, function, state). She then performed a complex priming study, in which she recorded the time taken to recognize individual lines from the program. Her interest lay in whether the line to be recognized was immediately preceded by a line from the same control structure, or from the same plan structure. The hypothesis was that if programmers form plan-based mental representations, then they should recognize lines faster when preceded by lines from the same plan structure.

Her results were quite clear cut and contrary to the plan-based view of expertise. Her subjects made fewer errors on the control-flow questions, compared with the data-flow and function questions and the effect of priming was consistently greater when the prime came from the same control structure. These results were replicated in a second experiment, using a longer program. However, in this second experiment an extra stage was added, in which the programmers were asked to make a modification to the program, and half were asked to provide verbal protocols while doing so. Pennington found that after the modification phase the dominant mental representation was of data flow and function, especially for those who had supplied protocols. Thus, Pennington concludes:

> While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organising principles for memory structures.
> (Pennington, 1987a, p. 327).

Although these two studies did not lead to the identification of particular strategies, it seems reasonable to include task characteristics as another determinant of comprehension process (given that they change the nature of the mental representation of a program).

Pennington (1987b) investigated the differences between the best and worst programmers (cf. novice-expert comparisons). From an initial programming test she was able to identify the top and bottom quartiles of a group of professional programmers, who were then closely examined during a comprehension task. Comparing the

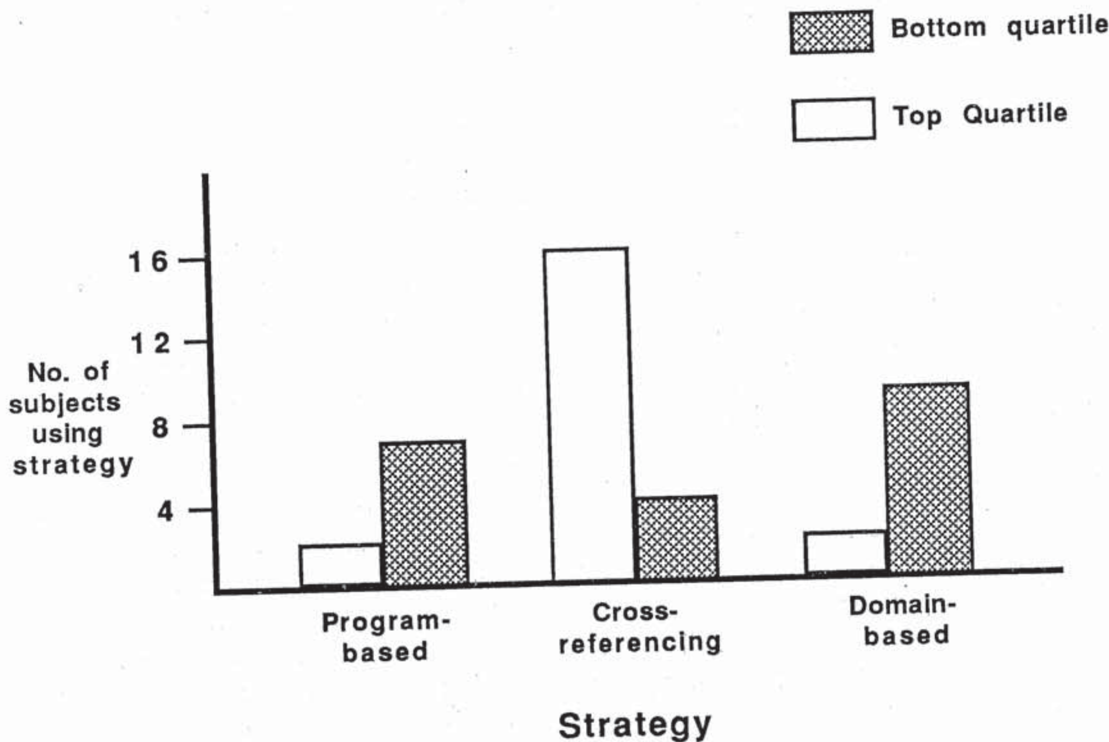

Figure 3: Pennington's (1987b) results revealing the different strategies used by her best and worst experts.

two groups of experts, she found that the most successful programmers adopted a *cross-referencing* strategy, considering not only the program, but also the real-world problem it addressed (Figure 3). This strategy can be contrasted with code-based and domain-based strategies, which were both commonly observed in the less successful experts.

In this result we can see that plan-based knowledge could be extremely important in the cross-referencing strategy, since plans (as described by Rist, 1986) provide bridges between domain knowledge and code fragments. However, it is quite possible that the less expert programmers possessed the same plan knowledge, but for some reason failed to use it.

From this review of studies about program comprehension, we can identify two important components of any strategy. Firstly there is the focus of attention, whether control flow, variables or maybe programming plans, and secondly, there is the scope of attention, which may be the code, the problem domain or both. Although not all of the combinations may actually occur in programming practice, the evidence suggests that a programmer's choice of strategy is influenced by his/her knowledge, the programming task, the program representation and the program's complexity.

4 Debugging strategies

A number of researchers have used the conventional novice-expert comparison for investigating debugging performance and some of these studies provide good examples of the need for a strategy approach. Using the implicit assumption that there

is only one debugging strategy, researchers conclude that differences between expert and novice debuggers must be due to differences in their knowledge about either the program or programming in general.

Gugerty and Olson (1986) found that expert programmers were both more likely to find bugs, and less likely to introduce new bugs than were novice programmers, despite the fact that both groups engaged in similar comprehension activities prior to debugging. They conclude that these differences must be due to differences in the comprehension ability of the two groups since the same set of activities led to differing amounts of comprehension. However, there are alternative explanations. Firstly, the activities observed by the experimenters might have been similar manifestations of quite different comprehension strategies, in which case the debugging differences will follow from comprehension differences.

A more-likely difference is that similar comprehension strategies led to very different levels of understanding about the program, which then forced the two groups to use quite different strategies for debugging. Faced with inadequate comprehension the novices had little choice but to make changes to the code, in the hope that they would either be right, or lead to improved comprehension. Gugerty and Olson (1986) describe a number of strategies used by their subjects, including simulated execution of the program (or part of it), working backwards from the observed error in output, elimination of parts (through partial testing) and adding print statements.

Vessey (1985, 1987) looked at the debugging performance of a number of expert Cobol programmers. Her methodology was complex, involving various classifications of the expertise of her programmers, and advanced statistical analysis. An initial test established the chunking abilities of her programmers, using a variant of the standard recall paradigm (cf. McKeithen *et al.*, 1981), abilities that can be equated with plan knowledge. In a subsequent debugging task she classified the strategies of her subjects into two types: one based on erratic behaviour and reinspections of parts of the program already covered and the other based on a fluid appproach to problem solving and a smooth progression through the program. It was then possible, using analysis of covariance, to establish whether the strategy used by programmers when debugging a program was more important than the knowledge structures possessed prior to examining the program. Chunking ability was able to account for 31% of the variation in debugging time, whereas debugging strategy could account for 74%. Vessey concludes

> the chunks programmers possess may not be important factors in debugging expertise, i.e. programmers may possess effective debugging strategies that are more important to the debugging process than the programming knowledge they possess.
> (Vessey, 1987, p. 70).

Thus, the evidence from studies of debugging reveals that a number of debugging strategies are available and that the success of debugging is as much a property of strategy choice as it is one of knowledge structures. Unlike the discussion of comprehension strategies above, there is no direct evidence about those factors that influence choice of debugging strategy. However, it is fair to label experience (and expertise) as one such factor, given the number of studies that show marked novice-expert differences. Other factors may include program authorship (Waddington, 1989) and debugging environment (Gilmore, 1990).

5 Studies of novices

Although the emphasis of this whole section is on the nature of *expert* programming knowledge and performance, knowledge-based theories of expertise are often used to draw conclusions about teaching, which is regarded simply as the passing on of the expert's knowledge, and novice difficulties are assumed to derive from the lack of this knowledge. Thus, a brief part of our discussion about the need for a strategy or process-oriented approach to programming must be an examination of some of the studies of novice programming, which reveal that novices often display strategic, rather than knowledge-based difficulties.

Perkins and Martin (1986) conducted a series of interviews with students using Basic. They describe how the main difficulties were 'fragile knowledge' and 'neglected strategies'. The former is knowledge that the student has, but fails to use when it is needed. Evidence that the student possessed the knowledge is not simply the fact that they had been taught it, but that on nearly 50% of occasions where hints had to be given to students, the student went on to solve the problem, even though the hint did not contain the appropriate knowledge.

A component of the difficulty seems to be that the students were not reading the program to discover what it actually did ('close tracking' or 'parsing'). This is an example of a 'neglected strategy', a general problem-solving strategy that should be, but is not, used. Mistaken strategies also exist, as can be seen in one of their students who appeared to be obsessed with using syntactic features taught recently, even when the problem only required simple structures taught and mastered some weeks before.

Perkins and Martin (1986, p. 225) summarize their novices' problems as 'fragile knowledge exacerbated by a shortfall in elementary problem-solving strategies', suggesting by way of conclusion that we should not view programming as an opportunity for the development of general problem-solving skills, since they are themselves required for successful programming.

White (1988) conducted a study with Prolog programmers, some of whom had previously learnt Pascal. Although it is not surprising that interference effects occurred, what is interesting is that Prolog novices were able to use the appropriate terminology (knowledge) for the Prolog at the same time as using an inappropriate strategy from Pascal.

Bonar and Cunningham (1988), in describing their use of the intelligent tutoring system Bridge (developed out of plan-based theories), comment on how their students were quite successful at developing an outline solution using plan-based concepts, suggesting that they had acquired and understood the contents of the programming plans. However, the bottleneck for the students came when trying to translate the plan-based outline into actual Pascal code:

> Matching between the Phase 2 output and Pascal code was problematic, however. Because there is not always a simple match between a plan component and Pascal code, students will sometimes make a reasonable selection that Bridge doesn't accept. (Bonar and Cunningham, 1988, p. 409).

Thus, it seems that having knowledge of the plans alone is inadequate, it is understanding how to use them that counts.

Finally, in some unpublished studies of POP11 novices, I made some observations that support the idea that novices can know plans, but not be able to use them

```
define listdouble(list);
var result;
Omitted initial value for result
until list = [] do
  result <> [^ 2*hd(list))] -> result;
  tl(list) -> list;
enduntil;
pr(result);
enddefine;
```

(a) Iterative solution for simple list processing problem, with common error.

```
define listdouble(list);
if list = [] then
  Omitted terminal value for function
else
  [^ 2*hd(list))] <> listdouble(tl(list))
endif;
enddefine;
```

(b) Recursive solution for same list processing problem, with analogous error.

Figure 4: Iterative and recursive plans, as used by novice, with same error made repeatedly.

successfully. The students had been taught about both iteration and recursion and were regularly given problems requiring them to code both types of solution. The problems used were comparable, except that students could choose between iterative or recursive solutions. In a number of cases the student's response to an error message was to switch from the iterative to the recursive 'plan', or vice versa. In one remarkable protocol a student wrote an iterative solution (Figure 4a), but omitted a necessary initialization (the only error). Rather than try to edit the program, he simply switched to a recursive solution (Figure 4b), making the analogous error (failure to return a value from the stopping condition). He then switched back to the iterative plan, and so on back and forth a total of five times (ten attempts!), before the mistake was detected. The next problem was similar but a little more complex. He wrote an almost successful iterative solution, but again omitted the initialization. The only evidence of learning from the previous problem was that it only required five attempts to find this same (!) mistake.

Thus, it seems that possessing knowledge is not the only problem that novice programmers have. In a number of cases they show that they have the knowledge, but also that they do not know how to adequately use it. It seems that programming may be rather like riding a bike, or some other motor skill, in that without practice it cannot be mastered.

6 Conclusions

The simplest conclusion from this survey of the nature of programming skills is that expertise is not as simple as we might sometimes think. Although high-level, efficient representations of programming knowledge develop with experience, it seems that this knowledge is not the sole determinant of programming success. Besides the chunking of knowledge structures, experts seem to acquire a collection of strategies for performing programming tasks, and these may determine success more than does the programmer's available knowledge.

A number of strategies for use in comprehension and debugging tasks have been described, and it seems that these enable experts to respond more effectively to unfamiliar situations, differing task characteristics and different language requirements. Finally, and importantly for teachers of programming, a number of studies of novices' difficulties have shown that they derive not only from lack of knowledge, but also from lack of strategy.

References

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Bellamy, R. K. E. and Gilmore, D. J. (1988). Programming plans: internal or external structures. *In* K. Gilhooly, M. Keane, R. Logie and G. Erdos (Eds), *Lines of Thinking: Reflections on the Psychology of Thought.* Chichester: Wiley.

Bonar, J. and Cunningham, M. (1988). *In* J. A. Self (Ed.), *Artificial Intelligence and Human Learning.* London: Chapman Hall.

Chase, W. G. and Simon, H. A. (1973). Perception in chess. *Cognitive Psychology,* **4,** 55-81.

Davies, S. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies,* **32,** 461-481

de Groot, A. D. (1965). *Thought and Choice in Chess.* The Hague: Mouton Press.

Gilmore, D. J. (1990). Models of debugging. Paper presented at *Psychology of Programming Interest Group: Second Workshop.* Walsall, January, 1990.

Gilmore, D. J. and Green, T. R. G. (1989). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology, HEP* **40(3),** 423-442.

Green, T. R. G., Bellamy, R. K. E. and Parker, J. (1987). Parsing-Gnisrap: A model of device use. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers,* vol. 2. Hillsdale, NJ: Ablex.

Gugerty, L. and Olson, G. M. (1986). Comprehension differences in skilled and novice programmers. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Kolodner, J. L. (1983). Towards an understanding of the role of experience in the evolution from novice to expert. *International Journal of Man-Machine Studies,* **19,** 497-518.

Laird, J. E., Newell, A. and Rosenbloom, P. S. (1987). SOAR: an architecture for general intelligence. *Artificial Intelligence,* **33**, 1-64.

McCabe, Th. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering,* **2**, 308-320.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. and Hirtle, S. C. (1981). Knowledge organisation and skill differences in computer programmers. *Cognitive Psychology,* **13**, 307-325.

Pennington, N. (1987a). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology,* **19**, 295-341.

Pennington, N. (1987b). Comprehension strategies in programming. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers,* vol. 2. Hillsdale, NJ: Ablex.

Perkins, D. N. and Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Hillsdale, NJ: Ablex.

Rist, R. (1986). Programming plans: definition, demonstration and development. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Hillsdale, NJ: Ablex.

Shneiderman, B. (1976). Exploratory experiments in programmer behaviour. *International Journal of Computer and Information Science,* **5**, 123-143.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies,* **23**, 459-494.

Vessey, I. (1987). On matching programmers' chunks with program structures: An empirical investigation. *International Journal of Man-Machine Studies,* **27**, 65-89.

Waddington, R. (1989). Unpublished Ph.D. Thesis, University of Nottingham, 1989.

White, R. (1988). Effects of Pascal knowledge on novice Prolog programmers. *Proceedings of the International Conference on Thinking.* Aberdeen.

Widowski, D. (1987). Reading, comprehending and recalling computer programs as a function of expertise. *Proceedings of CERCLE Workshop on Complex Learning.* Grange-over-Sands.