SOFTWARE ERGONOMICS
ergonomics
software
psychology
programming

# Psychological study of programming activity: a review

Jean-Michel HOC

Laboratoire de Psychologie du Travail de l'E.P.H.E., Equipe de Recherche Associée au CNRS, 41, rue Gay-Lassac, 75005 Paris, France

In his research, Jean-Michel Hoc, who is a Chargé de Recherche at the CNRS, has worked on problem resolution (notably in the design field) in professional situations. After working on issues associated with programmer training, he was moved on to consider programming strategies used by professionals, and programming assistance tools.

## COMMENTARY

Even though much recent research has been devoted to computing techniques *per se* (programming languages, programming methodologies, system development and the like), there has been less work done on the topic of human behaviour with respect to programming. Jean-Michel Hoc's paper surveys and comments on the main results obtained on the programmer's 'cognitive' activity. This study goes beyond a simple historical presentation by using instead a logical classification which gives a good account of the various fields which have been explored up to now: correction of errors in programs, programming language evaluation, teaching methodologies, etc. Above all, the paper starts a methodological discussion, in which the author criticises the overly use of inferential statistical tests performed without any reference to an underlying scientific psychological theory. The paper should make programmers, as well as psychologists who are interested in programming, aware of the necessity for deeper analyses of program design and understanding strategies. Only if such an awareness develops will it be possible to perform serious human-engineering (ergonomic) studies of the programming process, which would yield a better understanding of methods and techniques for the construction of programs.

**J.-P. Finance**

## CONTENTS

## Introduction

By the time Weinberg [Weinberg 71] published 'The psychology of computer programming' there had been very little research into the psychology of programming. Such work as had been done was largely orientated towards the selection of programmers. But the construction of a battery of tests does not require a profound knowledge of the target activity. For the battery to be valid it is only necessary for performance in the tests to correlate satisfactorily with performance in the particular activity, although the correlation itself may not readily be explained. Weinberg therefore was limited to suggesting research possibilities and giving methodological advice. Thus he insisted on the crucial role of teaching programming and the need for further studies in this field. He also recommended, for example, that we should not overlook the importance of openly observing programmers' behaviour with a view to determining how they operate, that experimental situations should not be too far removed from normal work situations (which generally imply work in a group) and that the study of the impact of programming aids and programming languages should not relate only to beginners.

Since then interest in the ergonomics of programming has grown rapidly among research workers, particularly as a result of the inversion of relative costs of hardware and software, the latter having increased considerably. However this renewed interest has been shown mainly by data-processing specialists rather than by psychologists. For instance the recent book [Shneiderman 80] 'Software psychology', in principle a survey of the results of research with a view to their application, primarily seeks to encourage further research in this field.

Work on this question has not on the whole followed Weinberg's recommendations, so that some thought needs to be given to the question of methodology and theory. This has already been undertaken by [Brooks 80], [Moher 82] and, in an excellent critical review [Sheil 81]. All these authors express the hope that future research will observe greater methodological rigour. But as Sheil remarks, the studies he described were mainly concerned with computer technology rather than with the strictly psychological approach. The methodological criticisms therefore seem insufficient to evaluate the results: discussion must take into consideration the relevance of the theoretical framework, whether or not this is made explicit in the research.

Plainly, an analysis of the activity of computer programming must be examined from two angles — in relation to data processing, of course, but also in relation to cognitive psychology. This we shall now attempt to do in presenting a critical review of these studies. It will be grouped under three headings: analysis of the programmer's work, study of *training* and evaluation of programming *aids*. Most research so far has been limited to the analysis of error-correcting activities and evaluation of programming languages.

At the end of this review methodological and theoretical aspects of this type of research will be discussed. Only those studies which include an analysis of observations will be quoted.

## 1. Analysis of the programmer's work

The programmer's activity can be divided into a certain number of sub-tasks, the main one of course being the construction of programs. So far not enough research has related directly to this. On the other hand there is a great deal of information available about error detection and correction. Other subsidiary tasks have been studied, such as comprehension, memorization and program modification, but these have been studied not so much for themselves but with a view to evaluating programming languages and programming aids.

### 1.1. Error detection and correction

Research in this field falls into two categories: studies of programs designed by the subjects themselves, and studies of incorrect programs not designed by those subjects.

In both cases authors generally adopted two forms of error classification: a first, approximate classification by depth, e.g. typing error, error of syntax, semantic error and finally error of logic; and a classification by type of incorrect instruction. This too may be interpreted in terms of depth, but in more detail.

#### 1.1.1. *Programs designed by the subjects*

The most complete study of the relative importance of each type of error is that carried out by [Youngs 74]. Using the first classification mentioned above he compares two distributions of errors: those made by beginners and those made by experienced subjects. In both groups errors of syntax (and typing errors) are fairly infrequent. This result is confirmed by Boies and Gould [Boies 74] who show, in studying the results of multiple runs of the same programs, that errors of syntax occur where there has been an insertion. Youngs finds that beginners more often make semantic errors than errors of logic, while with experienced subjects the reverse is observed. This result is compatible with the idea that beginners have difficulty in expressing algorithms correctly (semantic errors). For this reason errors of logic are not as common as in the case of experienced subjects who have outgrown these difficulties.

In examining the distribution of errors by type of statement the author observes that errors are most frequently found in declarative statements, assignments, loops and formats. Nagy and Pennebaker [Nagy 74] find a similar result for beginners, except that conditional statements replace loops in the group of most frequent errors (loops perhaps being less used by beginners).

While it may be useful to know that errors of syntax, detected automatically by machine, are not very frequent even for beginners, the conclusions of these studies are of limited scope. Once it has been established that assignment errors are fairly frequent a more detailed analysis of the errors is required. This can be achieved only by more direct study of *program design* strategies, which will allow errors to be interpreted according to psychological rather than data-processing classifications [Hoc 77, 78a, 81].

#### 1.1.2. *Programs not designed by the subjects*

The study of error detection strategies in programs not designed by the subject shows that experienced subjects tend to adopt an order of procedure. Gould and Drongowski [Gould 74] note that subjects correct the most superficial errors first of all (loops not closed, for instance), and only then concern themselves with deeper errors such as assignment errors. In asking experienced subjects to detect an error in programs which they did not write, [Gould 75] shows that assignment errors are the most difficult to detect. With regard to errors of logic [Michard 75] undertook a detailed analysis of several protocols. It appears that each subject *sets up* a model of what he supposes to be a correct program in order to *compare* it with the incorrect program. The setting up and comparison are carried out in a certain order: first of all at overall program plan level and then in detail if the error has not been detected. Thus Atwood and Ramsey [Atwood 78] rightly suggest that difficulty in correcting an error bears a direct relation to the level of detail needed in order to understand the program up to the incorrect element.

In the experiment quoted [Gould 74] the authors evaluate various types of error detection and correction aid. A control group received only the listing; the other groups had, in addition:

— either correct inputs and incorrect outputs;

— or correct inputs but outputs of which some were correct and some incorrect;

— or an indication of the type of error;

— or the line number of the error.

The aids studied do not seem very useful apart from the fact that the group which received only incorrect outputs obtained the worst results. The strategies of this group were very selective: in concentrating on certain well-defined parts of the program the subjects probably did not seek to understand the whole of the algorithm.

The hierarchization of error detection often observed seems to relate closely to different levels of program comprehension. We need therefore to study the mechanisms of program comprehension since this is used both in error detection and in program modification.

As a general rule the approach adopted in these studies makes psychological evaluation of errors difficult. Present classifications are still based too much on symptoms and not sufficiently on causes, which must be sought by analysing the underlying psychological mechanisms. It is of course easier to classify errors according to data-processing criteria, but the results have shown us the limitations of this method. This is simply a first, necessary stage; further research is then called for.

## 1.2. Program writing

As far as we know the first analysis of programming strategies was carried out by Rouanet and Gateau [Rouanet 67]. Even at a crude level of analysis, the authors noticed a considerable variation in the structure of constant problem programs written by subjects who were experienced (in management). The overall plan of the algorithm was, moreover, frequently masked by the effect of restrictions specific to the machine (1). The authors suggested that the problem should be analysed independently of the machine, concentrating on the circulation of information to be processed. This principle forms part of the basis of the LCP programming method [Warnier 75].

A very detailed analysis of the programming strategies of experts (in FORTRAN) was carried out by [Brooks 77]. Using the methodology developed by [Newell 72] (2), which had now become a standard work in the study of problem resolution, Brooks constructed a model of the procedure by which the algorithm is coded in programming language. This procedure forms part of a wider strategy, which includes comprehension of the problem and the search for a hierarchized program plan. The coding procedure model mainly uses two types of mechanism, each involving a dynamic model of the program: planning, which ensures overall control and the attainment of objectives in a predetermined order; and the *generation of statements* by a process of mental execution of the program.

Such studies are too few and probably too specific to enable us to understand the complexity of the problem-solving strategies involved in programming. A more detailed analysis of Brooks' work shows that these strategies involve interaction between basic mechanisms and the type of problem the subject is set. This is shown, for instance, in the opposition between a prospective mechanism, which generates statements in the order of their execution, and a retrospective mechanism, which generates in the reverse order. A recent experiment [Hoc 81] on advanced students shows that the extent to which the prospective mechanism predominates over the retrospective depends on the difficulty of the problem.

It therefore appears necessary to define program writing mechanisms clearly as basic components of strategies, in order to study the conditions in which they operate and form part of specific strategies. Among these conditions an important factor is probably the priorities set by the programmer to govern conflicting objectives such as minimization of memory occupation, calculation time, number of statements, writing time and so on; this hierarchy must be closely observed, as is shown by Weinberg and Schulman's experiments [Weinberg 74]. A basic component of strategies, which should be the subject of particular study when programming methods allow, is planning [Hoc 79], i.e. construction of the program from superstructures down to substructures (cf. 3.3. below), and its operating conditions.

On this latter point we are still restricted to working hypotheses — the result of studies carried out on beginners — quoted below. Planning seems mainly to relate to transferring structures of known programs. We must therefore undertake further research on the organization of knowledge in experienced programmers. At present only very limited information is available on the memorization of programs; this is presented below.

## 1.3. Other sub-tasks

With regard to memorization it will be useful to quote the experiments in which beginners and experienced subjects were required to memorize a program, the instructions being presented sometimes in order, sometimes out of order [Shneiderman 76, McKeithen 81]. They gave the same result. In the first case (in order) the experienced subjects memorized much more clearly than the beginners; in the second (out of order), equally badly. This result is generally interpreted by the authors in the same way as the results of the princeps experiment in chess [Chase 73] (3). Experienced subjects assimilate the program through high level configurations, which cannot be used if the program is out of order. In the experiment quoted, McKeithen *et al.* also attempted to throw some light on the organization of Algol concepts in subjects in the two groups. But it will be necessary to carry out studies on the organization of program memory, and not simply on the organization of such general conceptual knowledge.

[Michard 78] instructed experienced subjects to memorize a program, varying the form of presentation (FORTRAN listing vs flowchart) and retention time (5 min vs 24 h); subjects were then required to detect a logical error in a presentation of the program that had subsequently been modified. This work confirms a hyphothesis, now well established in respect of the memorization of texts, that there is separate syntactic and

(1) The programming language used was an assembler language.

(2) Subjects are instructed to 'think aloud' while writing their programs. From this the underlying psychological process may be inferred and a model set up by a production system; the outputs of this system are compared with the subject's behaviour in order to validate the model.

(3) For the game of chess, subjects (beginners and experienced) are presented with a position taken from a real game and a random position, which cannot be reached in the course of a game.

semantic coding, the first fading more quickly than the second. When the incorrect version of the program is presented in the short term, a comparison with the correct program as memorized at the syntactic level, is followed by a semantic comparison — a procedure which is a waste of time. The syntactic comparison is more likely when the first representation is in the form of FORTRAN listing, and less likely when it is in the form of a flowchart.

Very few results are available on the mechanisms of program comprehension. But an experiment by [Shneiderman 77a] shows that quality of memorizing is a good indicator of program comprehension.

All these analyses of the programmer's work show how the various sub-tasks of which it is composed are linked with each other; studies centred on some sub-tasks have some bearing on those dealing with others. Further research on programming strategies is probably the best way to show these links and to show how the various components of the programmer's activity and skill are integrated with each other.

## 2. Training

While programming may share the general properties of other problem-solving activities, it differs from them in that it requires the subject to exteriorize the *procedures* he is setting up.

It will be useful to contrast these programming situations with situations which have been termed 'result productions' [Hoc 78b], thus defining two opposite poles and assuming a continuum between them. In result production situations the aim of setting up a procedure lies simply in its execution. Setting up may take place at the same time as execution, as the particular data become available, without there being a need to make the procedure, and in particular its control structure, explicit. Programming situations require from the subject a higher level of awareness than result production situations; the latter however are more common.

Miller has studied the abilities of non-programmers with regard to this activity of exteriorization. In the first experiment [Miller 74a] he shows that some of the problems of expression of conditional structures are linked with the difficulty, already well-known in psychology, of manipulating logical disjunctive (OR) and negative (NOT) expressions in relation to conjunctive expressions (AND). In a second experiment [Miller 74b] where subjects must express a management procedure in their natural language, he confirms the difficulty found in expressing conditional structures: the resulting programs have a linear structure. Moreover he remarks on the incompleteness of specifications and the need to refer to the context in order to complete the programs.

We have examined the programming strategies of beginners in several experiments [Hoc 77, 78a, 78b] by observing the writing of management programs and taking note of errors and their corrections. We find that two essential requirements of computer programming are not easily met by the beginner. The first is the need to construct a mental representation of the program in order to exteriorize it (and thus to become aware of it); before achieving this structure, the subject expresses mental program operations which are sometimes very specific. The second is the need to design a procedure compatible with the functioning of the computer (at the level of definition of the language used). Until he has mastered this functioning and its consequences for program structure, the subject tends to transfer familiar procedures whose structure has been established on processing devices compatible with the computer. These incompatibilities become evident particularly in the data access and result production modes [Hoc 78a].

Several experiments by [Mayer, 75, 76, 81] show the advantage to teaching of having a model computer in order to learn programming. The subjects who benefit most from this concrete advantage, however, are those who are least good at mathematics. The author interprets these results and comments that the role of the model is to help the students to link new knowledge to previous knowledge. But the results obtained show also that exclusively practical knowledge is not enough to enable the programmer to learn to construct abstract algorithms, even though many aspects of programming employ mechanisms which can be shown in concrete form.

Thus we should not lose sight of the fact that computer programming is not a simple matter of the exteriorization of procedures. These must be set up for the purpose of operating a particular programming device. These studies of the learning of programming lead us to think that programmers should learn at an early stage how the computer functions. As is shown by various studies of industrial psychology in other fields, it is preferable to describe this functioning by relating it to instructions in the programming language used and not to an assembler language or a machine language [Du Boulay 81]. The effects of the computer's functioning on the design of program plans should be studied at the time when top-down programming methods are being taught [Hoc 77, 78a, 80; Kolmayer 79].

## 3. Evaluations

### 3.1. Evaluation of programming languages

In the main, the evaluation of languages has been concerned with conditional structures — a useful choice since, as we have seen, these present a certain difficulty. Most of the work has been done by a group of English research workers. In a first experiment on beginners, they showed first of all that easily nested structures (of the type IF — THEN — ELSE) gave better results than conditional branching (of the type IF — GO TO) in a program-writing task [Sime 73]. In a second experiment [Sime 77a] of the same type, the authors added to their evaluation of variation of the IF — THEN — ELSE expressed as 'IF condition — NOT condition — END condition'. This latter structure appeared to be better than the first two.

[Green 77] gave program comprehension tasks to experienced subjects and confirms these results for replies to retrospective questions of the kind 'with what type of data is this processing associated?', but finds no difference for prospective questions of the kind 'what processing will this type of data receive?'. By introducing structuring in the form of multiple choice rules (CASE), the same author [Green 80a] obtained no difference with nestable structures.

It is therefore probable that it is not the nestable character of the structure which facilitates its use, but rather the fact that the program can be more easily read.

The result obtained by [Shneiderman 76] with beginners in FORTRAN, according to which the arithmetic IF seems more difficult than the logical IF, can be interpreted from this point of view.

Moving away from strictly conditional structures, [Embley 78] proposes a new structure which would combine the CASE instruction and iteration. For comprehension tasks, the evaluation finds that it is superior to CASE or IF — THEN — ELSE instructions.

Several experiments by Gannon [Gannon 75, 76, 77] give information on other aspects of languages: for example the advantage of explicit use of types, and of regarding assignment as an instruction and not an operator.

The major disadvantage of these experiments is that they do not allow an unequivocal interpretation of results. Furthermore, if we observe interactions with this type of problem we find a considerable lessening of effect as subjects gain experience. Many questions remain outstanding and go beyond simple syntactical aspects of the languages [Green 80b; Jackson 80]: semantic problems must also be studied. Furthermore, the difficulty involved in comparisons between languages should not overlooked [Green 78]: if a structure is modified, for instance, the length of the program is also modified.

### 3.2. Evaluation of programming aids

Evaluations of programming aids are affected to a varying degree according to whether users are professionally experienced, the extent to which the aids in question are in current use, the type of task, and so on. It is difficult to draw sound conclusions from these rather random experiments.

The evaluation of the use of flowcharts provides a typical example. In an experiment on error detection, Brooke and Duncan [Brooke 80] came across two factors: listing against flowchart and structured ($^4$) or unstructured presentation. According to the type of performance evaluation (e.g. frequency of detection, time taken for detection, etc.) and the type of error, the effects of the various factors may disappear or be reversed. [Shneiderman 77a] finds almost no flowchart effect in program writing, modification and comprehension tasks.

As far as comments are concerned [Schneiderman 77a] has found that the performance of students modifying a program is better if the comments are high level. [Weissman 74] in a manual simulation program task finds that the task is carried out more quickly with the use of comments, but also that there are more errors.

Other studies on the use of tabulations on listings or of names of more or less significant variables leave us with a similar problem, the results being very isolated.

For such studies to provide interesting results from the point of view of research or application, it would no doubt be necessary to make a serious study of the interactions observed and not to attempt to draw conclusions that are too general. These interactions show clearly the complexity of programming as an activity. It will probably be necessary to seek specific aids for each of its aspects.

### 3.3. Evaluation of programming methods

While, at one time, learning to program was simply a matter of acquiring a language, the present aim is to train the student in a satisfactory method of programming. The methods taught are for the most part based on structured programming [Dahl 72]. They therefore call for the same type of planned approach in the writing of programs, by assembling and fitting together two basic plans: iterative structure and conditional structure. Planning, therefore, consists of expressing the superstructures before introducing substructures. As these methods have only recently come into use, evaluations have mainly related to beginners.

Authors already quoted with regard to the evaluation of conditional structures have evaluated a method of this type for the nesting of structures of the IF — THEN — ELSE type [Sime 77b]. In comparison with free expression, a situation in which subjects are required to express the whole of the conditional structure before inserting other structures of the same type appears better from the point of view of errors and their lifetime. In an experiment in which subjects did not have this planning task, but were merely not allowed to use GO TO, contradictory results were found: programming with branching appears easier, although the opposite effect was observed for the modification of programs [Lucas 76]. Furthermore, these results also contradict those obtained in the evaluations of conditional structures [Sime 73, 77a]. Lucas and Kaplan think that the advantages of structured programming over programming with unconditional branching cannot really show without sufficient training.

In program comprehension tasks contradictory results are found: structured programs are sometimes understood better [Sheppard 79], and sometimes equally well [Weissman 74]. These two authors find no effect as far as the correction of errors is concerned. But the former shows that a structured program is easier to modify. It is also found to be easier to memorize [Love 77].

These studies, however, deal with only one aspect of the evaluation of modern methods: structured programming, i.e. the use of its basic constructions. While Sime's experiment [77b] introduces a hierarchy into expression, it may be asked whether the subject is given this instruction as a method or simply as an aid to memorization while writing the program. Studies carried out in France give more direct information on the evaluation of this type of method, but only in the context of learning.

A longitudinal study [Hoc 78a] of the LCP method [Warnier 75] showed two basic difficulties encountered by students learning the method. Firstly, while the method dissociates structuring of data (and of results) from program structuring (which is deduced from the former), subjects experience great difficulty in making this dissociation. Furthermore, while structuring of data (and of results) is often incorrect, improved performance may be observed in the construction of the flowchart, for which students are able to construct the program by executing it mentally. Secondly, since subjects learn the method, errors of data (and result) structuring are often linked with insufficient knowledge of these rules.

Learning of the deductive method [Pair 79] has also been studied [Kolmayer 79], from the point of view of its two principal characteristics: the procedure is retrospective (the program is constructed from the results toward the data) and planned (in the sense already mentioned).

($^4$) i.e. using control structures of 'structured programming'.

The author carried out analyses of observations on several volunteers. These subjects found difficulty in following the retrospective procedure as soon as the processing became complex (this result is confirmed by [Hoc 81]). A possible explanation is that beginners have a tendency to execute the program mentally before writing it. The planning procedure appeared also to be abandoned in the same conditions and for the same reasons. As far as the overall plan of the program is concerned, it is often constructed before the subject has acquired all the relevant information regarding the initial statement: it is often constructed before the subject has acquired all the relevant information regarding the initial statement: it is therefore probably transferred from already known programs, and based on very general information.

These methods revealed the subjects' programming strategies, in providing referential norms when they are assessed at a low level of experience: without them the subjects' strategies are not often explicit. Such study also shows clearly the difficulty of acquiring design procedures. But this cannot be shown in isolated experiments during which beginners have not sufficient time to familiarize themselves with the method.

Even if the necessary precautions are taken, however, this evaluation is insufficient. In particular it must also be conducted on experienced subjects, in order to determine whether these methods are equally valid at that level.

## 4. Methodological and theoretical discussion

We shall not reiterate here the separate evaluations undertaken for each of the sub-themes. Discussion will be at a more general level, without entering into the detail of criticisms already made [Lewis 81; Sheil 81; Moher 82], except for any which may seem to us to be in need of developing or explaining in greater detail.

After reviewing the present state of research into the ergonomics of programming, we find it difficult to draw unambiguous conclusions, as is done a little too quickly by [Shneiderman 80]. Most studies dispense with a psychological analysis of programming activity, making a superficial analysis of the task strictly from a data processing point of view; we have observed this particularly in work on the analysis of errors and on programming aids. The complexity of the activity involved does not lend itself easily to portrayal by the methodology followed. In order to realize this, it is only necessary to note the complex interactions observed and the difficulty of interpreting results.

Many experiments were aimed at obtaining generally applicable results. On this point they have already been criticized by the authors quoted: subjects were too often beginners, experimental situations too artificial, variables badly defined, etc. Some of these criticisms are worth giving in detail.

In the case of experiments which aim to be inductive, we may expect the general scope of the results to be stated. Use of tests of significance allows generalization so as to include a population of subjects of the same type as those in the experiment, but who are subject to the experimental situation itself. But while this first generalization is necessary, it does not go far enough. The kind of situation must also be defined.

On the first point (statistical inference), the relevance of tests of significance should be looked at more closely. While the latter may allow us to conclude that the effects examined exist, they allow no conclusions to be made with regard to the extent of these effects. But what interest can there be, particularly when it comes to application, in concluding that an effect exists if it is negligible? Tests of significance are therefore inappropriate and must be replaced by other inferential methods, such as fiducial methods [Lépine 75].

On the second point (general scope), before conclusions are drawn that are too general, the greatest attention should be paid to the interactions observed. For example, to conclude that flowcharts do not on the whole have any effect on performance would be to overlook hypothetical interactions such as that quoted by Moher and Schneider [Moher 82]: the more complex the program and the more macroscopic the flowchart, the more useful would be the aid. On this point, it is to be regretted that most experiments, in attempting to draw general conclusions too hastily, are too simple in structure to allow such interactions to be examined.

The results of these experiments are also expected to be applicable from the point of view of ergonomics. Their field of application may be defined as a function of the relationship between experimental situations and normal working ones. Although this relationship is not always made obvious we should not lose sight of the fact that an experimental situation is a reduction of the work situation to which it refers. It simplifies it; it structures it by taking a point of view which allows information, which would be lost in the complexity of the work situation, to be clearly distinguished.

The relevance of such a reduction can be established only by referring to a theoretical frame of reference within the activity concerned. But it is just such a frame of reference which is lacking here. Whether the authors are studying programming strategies, memorization activities, modification, comprehension or program correction, reference to psychological research is absent. It is true that the theoretical frames of reference of psychology are insufficiently developed, but they are more developed than the over-simplified implicit models of some of the studies quoted.

But where psychology is not able to supply models capable of establishing the relevance of experiments, we must turn to more open observation of the behaviour of programmers. Observation, even if it does not lead directly to results of general scope, is an integral part of experimental procedure. It can provide reasonable hypotheses and relevant questions which cannot be derived from theoretical assumptions. The psychology of problem solving certainly uses this method of observation, as may be seen in Newell and Simon [Newell 72].

In order to further this research we must no doubt take a closer look at design strategies, comprehension and program memorization. The need to find models of these activities should give stricter orientation to studies of activities as diverse as comprehension of statements or problem definition, program modification and correction, etc. These models should also orientate the design of aids together with their evaluation.

In this respect we must make a clear distinction between studies relating to beginners and those relating to experienced programmers. It is clear that the activity of programming is built up of numerous components and requires a prolonged period of learning. The expert's range of strategies is probably ([5]) not the same as that of the beginner. It would be useful to determine the basic components of this range; from this we might gain some interesting information on the design of teaching programs.

But we should also go beyond the rather over-simplified framework of experiments, for example by taking account of the design of complex programs, the heuristics of access to extended knowledge and the structuring of this knowledge. Analyses of data collected by open observation would no doubt be more useful than hurrying through experiments that may be premature.

## 5. Conclusion

It is a truism to say that ergonomics is an attempt to optimize socio-technical systems. But it is not always clear that this optimization cannot be effected without complete understanding of the two sub-systems: human and technical. This is shown by research into the ergonomics of programming: so far this research has adopted a technical point of view which has not allowed satisfactory results to be reached.

There is of course still too little information available on the characteristics of the human factor in question: the strategies of problem resolution. Furthermore the latter are no doubt greatly affected by the technical environment. The opposite extreme of psychologism must therefore also be avoided. For this reason, scientific collaboration between psychologists and data-processing specialists appears to be a necessary condition for fruitful research which does not neglect the pre-occupations of either side.

Beneath the problem of applied psychology posed by the study of computer programming, lie problems of fundamental psychology which cannot be solved in the laboratory. This is shown by recent developments in research on problem solving, which is now increasingly concerned with data collected in normal working situations. Thus, far from being simply a field of application of psychology, programming is also one of the situations from which we may obtain fundamental information about psychology.

We must therefore hope that data-processing specialists interested in the psychology of programming will not concentrate their efforts exclusively on experimental laboratory techniques, which represent only one stage in the experimental process. It is also to be hoped that more psychologists, including fundamentalists, will undertake research in this field.

([5]) One of the TSI referees asked me if this "probably" is necessary. I will attempt to reply. Does the activity of programming create in *homo sapiens* new strategies of problem solving? Will he thus become Superman? These are major questions which some have already answered in their inmost being. But, humble researcher in psychology that I am, I am wary of my inmost being, at the risk of appearing very empirical — like Descartes and many others — in the eyes of the great ones of our world (Papert, for example). To come back to more serious considerations, the question in general seems to me to be as follows: Does the expert have strategies at his disposal which the beginner does not? or can the expert employ in his field strategies which are available to the beginner, but which cannot be adopted without knowledge specific to the field (operating conditions) which, obviously, the beginner does not yet have? This is one of the questions which at present is directing my work on professional programmers.

## Comments on J. M. Hoc's article ([1])

1. Would it not have been possible to take a little more account of modern developments in programming? It is not only a question of 'structured programming' (which is not simply the use of restricted control structures!) but also of more recent concepts such as use of the modularity based on abstract types, 'object-orientated' programming, etc.

Are there any evaluations of the use of languages such as Smalltalk (cf. Cointe, TSI, Vol. 1, No. 4)? How do children react to S. Papert's Logo language and, more generally, what is the impact of languages manipulation of objects?

2. A question relating to the one above, but which concerns aids: the 'programming aids' mentioned by the author (flowcharts . . .) seem very out of date. Do we have results on the use of more modern aids, such as syntax editors (MENTOR, CPS, GANDALF), integrated programming environments, modern programming systems, the use of networks for cooperation between programmers, etc.? If J. M. Hoc's hope is realized and many psychologists start serious study of programming, they must not provoke a division between the disciplines of psychology and computer science by making use of outdated concepts.

([1]) Comments made by A. Michard (Inria, Sophia Antipolis) and B. Meyer (EDF).

## Author's reply (J. M. Hoc)

The referees show how far ergonomic evaluations are lagging behind research into computer programming. They imply that this state of things is due to psychologists' lack of training in computer science. A quick scan of the bibliography of this article, as I have remarked, shows that most of the ergonomics studies have been carried out by data-processing specialists, so that it is difficult to question their competence in this field.

Why then have 'out-dated' concepts been used? At least two reasons can be suggested:

(1) Even if these concepts appear outdated to the researcher, they may not be so in current programming practice and teaching, or not to the extent he thinks. Ergonomics studies naturally relate to real work situations and only with greater difficulty to those of the future.

(2) Similarly, it is difficult to find suitable subjects in order to evaluate aids which are as yet not widely available. In particular it is necessary to find subjects who are sufficiently familiar with the aids in question. Otherwise, as [Sheil 81] rightly remarks, there is a risk of giving too much importance to evanescent phenomena which disappear rapidly with practice.

Access to subjects is therefore a condition which it is not always easy to meet, even for a psychologist within a data-processing research organization, as is the case with one of my referees [A. Michard]. Moreoever, it is to be hoped that he and his colleagues will undertake wider research on programming, in view of their privileged situation of being in permanent contact with research in this field.

It is in fact to prevent psychologists from basing themselves on 'outdated' concepts that I suggest cooperation with data-processing specialists. On this point, it is good to see that an association of this type has been set up at European level, the first conference on the theme 'Cognitive Engineering' (Amsterdam, 10–13 August 1982) having brought together researchers in the two disciplines on the wider theme of the ergonomics of programming.

With regard to the work of S. Papert, it is true that several publications have been devoted to it. I have limited myself to a review of literature on 'professional' programming, deliberately leaving aside 'educative' programming. This latter subject poses other problems, since it is above all considered as a means of acquiring the concepts of other disciplines (mathematics, physics, etc.). A. Michard could present a more specific review on this theme, as a result of his current work on the question.

In general it is to be hoped that research will develop in both areas, and also on aids which make programming more readily available to users who are not specialized in data-processing. The stakes, as we know, are high.

## REFERENCES

[Atwood 78] M. E. Atwood and H. R. Ramsey: *Cognitive structures in the comprehension and memory of computer programs: an investigation of computer debugging;* Tr–78–A21, 1978, U.S. Army Research Institute for the Behavioral and Social Sciences, Alexandria (Va).

[Boies 74] S. J. Boies and J. D. Gould: *Syntactic errors in computer programming;* Human Factors, **16** (3), 253–257, 1974.

[Brooke 80] J. B. Brooke and K. D. Duncan: *Experimental studies of flowchart use at different stages of program debugging;* Ergonomics, **23** (11), 1057–1091, 1980.

[Brooks 77] R. E. Brooks: *Toward a theory of the cognitive processes in computer programming;* International Journal of Man-Machine Studies, **9** (6), 737–751, 1977.

[Brooks 80] R. E. Brooks: *Studying programmer behavior experimentally: the problem of proper methodology;* Communications of the ACM, **23** (4), 207–213, 1980.

[Du Boulay 81] B. du Boulay, T. O'Shea and J. Monk: *The black box inside the glass box: presenting computing concepts to novices;* International Journal of Man-Machine Studies, **14** (3), 237–249, 1981.

[Chase 73] W. G. Chase and H. A. Simon: *Perception in chess;* Cognitive Psychology, **4** (1), 55–81, 1973.

[Dahl 72] O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare: *Structured Programming;* 1972, Academic Press, London.

[Embley 78] D. W. Embley: *Empirical and formal language design applied to a unified control structure;* International Journal of Man-Machine Studies, **10**, 197–216, 1978.

[Gannon 76] J. D. Gannon: *An experiment for the evaluation of language features;* International Journal of Man-Machine Studies, **8**, 61–73, 1976.

[Gannon 77] J. D. Gannon: *An experimental evaluation of data type conventions;* Communications of the ACM, **20** (8), 584–595, 1977.

[Gannon 75] J. D. Gannon and J. J. Horning: *The impact of language design on the production of reliable software;* IEEE Transactions on Software Engineering, **SE–1**, 179–191, 1975.

[Gould 75] J. D. Gould: *Some psychological evidence on how people debug computer programs;* International Journal of Man-Machine Studies, **7**, 151–182, 1975.

[Gould 74] J. D. Gould and P. Drongowski: *An exploratory study of computer program debugging;* Human Factors, **16** (3), 258–277, 1974.

[Green 77] T. R. G. Green: *Conditional program statements and their comprehensibility to professional programmers;* Journal of Occupational Psychology, **50**, 93–109, 1977.

[Green 80a] T. R. G. Green: *Ifs and thens: is nesting just for the birds?* Software Practice and Experience, **10**, 1980.

[Green 80b] T. R. G. Green: *Programming as a cognitive activity;* In: Human Interaction with Computers; H. T. Smith and T. R. G. Green (Eds), 1980, 271–320, Academic Press, London.

[Green 78] T. R. G. Green, M. E. Sime and M. Fitter: *Thoughts on behavioral studies of programming;* Memo 211, 1978, MRC Social and Applied Psychology Unit, Sheffield.

[Hoc 77] J. M. Hoc: *Role of mental representation in learning a programming language;* International Journal of Man-Machine Studies, **9** (1), 87–105, 1977.

[Hoc 78a] J. M. Hoc: *Étude de la formation à une méthode de programmation informatique;* Le Travail Humain, **41** (1), 111–126, 1978.

[Hoc 78b] J. M. Hoc: *La Programmation informatique comme situation de résolution de problème;* Thesis, 1978, Laboratoire de Psychologie de Travail de l'E.P.H.E., Paris.

[Hoc 79] J. M. Hoc: *Le problème de la planification dans la construction d'un programme informatique;* Le Travail Humain, **42** (2), 245–260, 1979.

[Hoc 81] J. M. Hoc: *Planning and direction of problem-solving in structured programming: an empirical comparison between two methods;* International Journal of Man-Machine Studies, **15** (4), 363–383, 1981.

[Hoc 80] J. M. Hoc and A. Kerguelen: *Un exemple de dispositif informatique expérimental pour la psycho-pédagogie de la programmation;* Informatique et Sciences Humaines, **44**, 67–86, 1980.

[Jackson 80] M. Jackson: *The design and use of conventional programming language;* In: Human Interaction with Computers; H. T. Smith and T. R. G. Green (Eds), 321–347, 1980, Academic Press, London.

[Kolmayer 79] E. Kolmayer: *Développement et évaluation d'une méthode de programmation;* CRIN (Nancy), 79–R–074, 1979.

[Lépine 75] D. Lépine and H. Rouanet: *Introduction aux méthodes fiduciaires: inférence sur un contraste entre moyennes;* Cahiers de Psychologie, **18** (4), 193–218, 1975.

[Lewis 81] C. Lewis: *Computers, their users, and psychology* (a review of B. Shneiderman: Software Psychology; 1980); Contemporary Psychology, **26** (8), 602–603, 1981.

[Love 77] T. Love: *Relating individual differences in computer programming performance to human information processing abilities;* Ph. D. Dissertation, 1977, Washington.

[Lucas 76] H. C. Lucas and R. B. Kaplan: *A structured programming experiment;* The Computer Journal, **19** (2), 136–138, 1976.

[Mayer 75] R. E. MAYER: *Different problem-solving competencies established in learning computer programming with and without meaningful models;* Journal of Educational Psychology, **67** (6), 725–734, 1975.

[Mayer 76] R. E. MAYER: *Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order;* Journal of Educational Psychology, **68** (2), 143–150, 1976.

[Mayer 81] R. E. MAYER: *The psychology of how novices learn computer programming;* ACM Computing Surveys, **13** (1), 121–141, 1981.

[McKeithen 81] K. B. MCKEITHEN, J. S. REITMAN, H. H. RUETER and S. C. HIRTLE: *Knowledge organization and skill differences in computer programmers;* Cognitive Psychology, **13** (3), 307–325, 1981.

[Michard 75] A. MICHARD: *Analyse du travail de diagnostic d'erreurs logiques dans un programme FORTRAN;* INRIA, Paris. CO7602–R48, 1975.

[Michard 78] A. MICHARD: *Représentations opératoires et modèles du processus dans les tâches de diagnostic;* Thesis, 1978, INRIA, Paris.

[Miller 74a] L. A. MILLER: *Programming by non-programmers;* International Journal of Man-Machine Studies, **6**, 237–260, 1974.

[Miller 74b] L. A. MILLER and C. A. BECKER: *Programming in natural English;* IBM RC5137, 1974, T. J. Watson Research Center.

[Moher 82] T. MOHER and G. M. SCHNEIDER: *Methodology and experimental research in software engineering;* International Journal of Man-Machine Studies, **16** (1), 65–87, 1982.

[Nagy 74] G. NAGY and M. C. PENNEBAKER: *A step toward automatic analysis of student programming errors in a batch environment;* International Journal of Man-Machine Studies, **6**, 563–578, 1974.

[Newell 72] A. NEWELL and H. A. SIMON: *Human Problem Solving,* 1972, Prentice Hall, Englewood-Cliffs, N.J.

[Pair 79] C. PAIR: *La construction des programmes;* R.A.I.R.O. Informatique, **13** (2), 113–137, 1979.

[Rouanet 67] J. ROUANET and Y. GATEAU: *Le travail du programmeur de gestion: essai de description;* 1967, AFPA-CERP, Paris.

[Sheil 81] B. A. SHEIL: *The psychological study of programming;* Computing Surveys, **13** (1), 101–120, 1981.

[Sheppard 79] S. B. SHEPPARD, B. CURTIS, P. MILLIMAN and T. LOVE: *Modern coding practices and programmer performance;* IEEE Transactions on Software Engineering, 41–49, Dec. 1979.

[Shneiderman 76] B. SHNEIDERMAN: *Exploratory experiments in programmer behavior;* International Journal of Computer and Information Sciences, **5** (2), 123–143, 1976.

[Shneiderman 77a] B. SHNEIDERMAN: *Measuring computer program quality and comprehension;* International Journal of Man-Machine Studies, **9** (4), 465–478, 1977.

[Shneiderman 77b] B. SHNEIDERMAN, R. E. MAYER, D. MCKAY and P. HELLER: *Experimental investigations of the utility of detailed flowcharts in programming;* Communications of the ACM, **20** (6), 373–381, 1977.

[Shneiderman 80] B. SHNEIDERMAN: *Software Psychology: Human Factors in Computer and Information Systems;* 1980, Winthrop, Cambridge, Mass.

[Sime 73] M. E. SIME, T. R. G. GREEN and D. J. GUEST: *Psychological evaluation of two conditional constructions used in computer languages;* International Journal of Man-Machine Studies, **5**, 105–113, 1973.

[Sime 77a] M. E. SIME, T. R. G. GREEN and D. J. GUEST: *Scope marking in computer conditionals: a psychological evaluation;* International Journal of Man-Machine Studies, **9** (1), 107–118, 1977.

[Sime 77b] M. E. SIME, A. T. ARBLASTER and T. R. G. GREEN: *Reducing programming errors in nested conditionals by prescribing a writing procedure;* International Journal of Man-Machine Studies, **9** (1), 119–126, 1977.

[Warnier 75] J. D. WARNIER: *Les Procédures de Traitement et leurs Données (LCP);* 1975, Editions d'Organisation, Paris.

[Weinberg 71] G. M. WEINBERG: *The Psychology of Computer Programming,* 1971, Van Nostrand, New York.

[Weinberg 74] G. M. WEINBERG and E. L. SCHULMAN: *Goals and performance in computer programming;* Human Factors, **16** (1), 70–77, 1974.

[Weissman 74] L. WEISSMAN: *Psychological complexity of computer programs: and experimental methodology;* ACM SIGPLAN Notices, **9**, 1974.

[Youngs 74] E. A. YOUNGS: *Human errors in programming;* International Journal of Man-Machine Studies, **6**, 361–376, 1974.