1. *Historical Background*

The study of the psychology of programming started in the 1970s[a]. Two distinct periods can be identified. In the first, research was largely in the hands of computer scientists and its main purpose was the evaluation of software tools in terms of performance. This was the period of experimental studies looking to analyse the effect of different factors (such as indentation) on performance in different programming tasks, without cognitive models to take account of the activities in the different tasks. The book that best illustrates the studies of this period is that by Shneiderman (1980); it had some impact on the world of computing when it was first published.

For the last 15 years or so there has been a growing interest in the field on the part of psychologists and ergonomists, who have seen programming as a field in which to study the activities of design, comprehension, and expert problem solving, as well as a means of developing and evaluating tools to help with them. This second period is characterised by the development of cognitive models of programming and by the use of more clinical methods of activity analysis, alongside of, and as a complement to, experimental methods. Researchers in this field of science meet at annual or biannual workshops: that of the Psychology of Programming Interest Group in Europe, and the Empirical Studies of Programmers in North America. These meetings have scientifically been very productive.

1.1 The 1970s

1.1.1 The Experimental Paradigm

The roots of the study of programming lay in computer scientists' interest in testing new programming tools. As programming itself evolved by distancing itself from the use of machine languages, so methods, languages and visualization tools were developed. The need to validate these tools experimentally thus arose. To do this, computer scientists borrowed methods from experimental psychology to evaluate the tools[1]. These experimental methods allow one to test the effect of one or more

[a]Hoc *et al.* (1990) place the birth in the 1960s. It is true that a French study (Rouanet and Gateau, 1967) of that period aimed to analyse data processing, but this study was completely unique at the time and we have to wait until the 1970s before further work appears.

factors, called independent variables, on the values taken by a dependent variable. The principle is to construct an experimental situation by varying the values of one independent variable. In the simplest case, we have an experimental condition where the variable is present and a control condition where this variable is absent, all other things being the same. One then measures the effect of this variable on the dependent variable, which is, in these studies, usually a performance indicator.

To illustrate this approach, we cite a hypothetical example given by Shneiderman (1980). This is a bi-factorial study, that is, a study involving two factors. The two independent variables whose effect we want to measure are, on the one hand, the use of meaningful mnemonic variable names and, on the other, the use of comments. Each variable may be present or absent. The dependent variable is a measure of performance in modifying a program. Combining these two factors, we obtain four experimental conditions to which one can randomly expose the experimental subjects. It is important that the factor of the subjects' experience is properly controlled in the study, which means that one must be sure that the subjects come from a single population whose characteristics can be described. Table 1.1 shows the hypothetical results of such an experiment.

From the means shown in the last column and the last row, one can see that there is an effect from the factor 'presence of comments' on the one hand, and from the factor 'presence of mnemonic names for variables' on the other.

A second stage is then to carry out statistical tests, of the analysis of variance type (ANOVA), which will allow us to show what is the probability of not finding these results when the experiment is replicated with similar groups. If the probability is small, say one in a thousand for the effect of an independent variable, then one can say that this variable has a significant effect at p = 0.001.

As stated above, the dependent variables most commonly used in the studies of this period are performance indicators. These indicators are often related to different programming tasks, for example, in a debugging task, they might be the number of errors detected and corrected, and the time taken to detect them. These indicators measure very clearly success in a given task in terms of the final product of the activity but don't in any sense describe the activity used to accomplish the task. We shall see that this causes a problem in interpreting the results.

Another experimental approach that characterizes the studies of this first period is the search for correlations between, on the one hand, factors belonging to the subject, e.g. the number of years of programming experience, and, on the other

Table 1.1 Mean of the modification scores obtained in a bi-factorial experiment (taken from Shneidermann, 1980, p. 20).

	Variable names		
	Mnemonic	Non-mnemonic	
With comments	84.5	72.3	78.4
Without comments	75.4	56.8	66.1
	80.0	64.6	72.3

hand, behavioural factors, e.g. success in a test supposed to measure the ability to understand programs. The practical objective of such studies was to develop tests of competence for staff selection purposes. It is important to remember, however, that the establishment of a correlation between two factors does not mean that there exists a causal relationship between them.

1.1.2 Methodological Criticism

A number of methodological criticisms can be made of this early work. In practice, it is difficult to isolate a single factor and vary it without inducing other changes in the situation. One can still try to isolate it but at the risk of creating a rather artificial situation. We shall illustrate this methodological limitation through two examples, the first demonstrating, to our way of thinking, a rather amateurish experimental approach, and the second illustrating a more professional approach, but one that creates a relatively artificial situation.

Gannon (1977) studied the effect of typing in a programming language on the performance of a programming design task as measured *inter alia* by the number of errors in the program. The independent variable has two values: 'static typing present' (that is, the type of a variable must be declared before it is used) and 'static typing absent'. In order to change this variable, the author has had to use two different languages. This has the effect of varying not only the factor studied but also many other syntactic and semantic characteristics of the languages. Interpretation of the results is thus difficult because there are other factors, confused with the independent variable, which could also be the cause of the results obtained.

To avoid this methodological problem, some researchers[2] have created micro-languages that vary only in one or a few well-identified factors. They have thus been able to compare the effect of different types of loops (goto, nested, etc.) or different types of test on the performance of different tasks. In doing this, they create a rather artificial situation in comparison with the situation in which programming usually takes place. They defend their position by stating

> The technique of using severely restricted micro-languages allows comprehensive conclusions to be reached . . . In our opinion, at present more can be lost than learned by increasing the size of the experimental language[3].

Gannon defends the opposite approach, considering that the use of micro-languages is too artificial:

> Language features must be evaluated in the context in which they are used, and creating too small a language may prevent the observation of errors resulting from the interaction of language features[4].

1.1.3 Absence of a Theoretical Framework

Another problem is linked to the absence of a theoretical framework and to the use of performance indicators that measure the final result of the task; this makes it difficult to explain the mechanism of an effect. In fact, these studies offer no explanatory model in terms of cognitive processes for how an effect works. In no case do they explain how and why a variable has an effect on the performance. Further, the absence of an effect on performance has, in our view, very limited value in indicating the effect of the factor under consideration. This effect may exist but just not be detected by the final indicators. In order to detect the effect in such a case we need indicators that bear on the individual activities themselves rather than on the task as a whole. It was only in research of the second period that methods were introduced to analyse the activity in its true sense. We shall give an example that illustrates the methodological limitations of the first period and look forward to the paradigmatic changes that came into use during the second period.

The example again concerns the use of meaningful or non-meaningful names for variables. Recall that a program is made up of predefined symbols (operators, reserved words) and of names chosen by the programmer (names of variables, procedures, etc.). One might expect that the presence of meaningful names in a program will help in understanding it. However, early studies relating to this factor do not always reveal that it has a positive effect on understanding. Weissman (1974) failed to find that the factor had any effect on different programming tasks. Another study (Sheppard, Curtis, Milliman and Love, 1979) failed to find any effect on the ability to reproduce a functionally equivalent program from memory. Shneiderman (1980) showed that the more complex the program, the more the use of mnemonic variable names helps understanding, measured by the error detection rate in a debugging task. Although some of this work allows us to understand better under what conditions an effect of this variable can be observed in terms of comprehension performance, it does not show what cognitive mechanisms are at work.

Researchers studying program understanding in the second period built themselves theoretical frameworks and corresponding experimental paradigms. Thus one might hypothesize that a meaningful name is a semantic index that allows for a knowledge structure (in this case a variable schema) stored in long-term memory to be activated and for inferences to be made on the basis of the knowledge retrieved. One study[5] allowed the effect of variable names on understanding to be grasped using a paradigm pertinent for studying the inferences made when seeking to understand a program: a completion task. If the statement deleted is 'Count := 0' and the 'counter schema' is recalled thanks to the name of the variable in the update statement 'Count := Count + 1', experts will be able to infer the missing statement more easily than novices. In this study, therefore, it is clear that the name is a semantic index that allows a schema to be activated, which makes it possible to infer the missing part in the schema.

According to this explanatory model of the effect of variable names, we can understand why a performance indicator referring to the outcome of the task cannot measure such an effect. These activation mechanisms are transitory and the processes of comprehension are based on a multiplicity of input data, not merely the meaningful variable names. Thus in Sheppard's experiment, the failure to detect any effect might be due to the fact that the programs were annotated; the effect of the comments might have been to mask, if not to nullify, the effect of the meaningful variable names.

The difficulty of interpreting the absence of an effect, as much as its presence, and the contradictory results observed among the early studies is underlined by Hoc *et al.* (1990). They illustrate it by reference to studies on the effect of various notational structures such as the flowcharts and listings. The results are contradictory: some authors[6] claiming to find a performance improvement with the use of flowcharts, while others[7] found an absence of any effect. We have to wait for later studies to understand which components of the task are affected by the use of flowcharts. Thus Gilmore and Smith (1984) emphasize the fact that the assistance that a given notational structure provides is related to the strategy that the programmers follow to accomplish a task. For a debugging task, the subjects observed followed either a strategy of program execution or a strategy of understanding which allows them to build a mental model of the program. The flowchart, however, is of assistance specifically in the process of execution; the subjects who follow this strategy perform better with the flowcharts than with the listing.

1.1.4 Badly Used Theoretical Borrowings

We have emphasized above the lack of a theoretical framework, which is characteristic of early work. We must recognize, however, that some theoretical ideas were borrowed from psychology but that these ideas were very badly used by computer scientists. The most illuminating example is the borrowing of the notion of short-term memory. A classical distinction in psychology, albeit the subject of a long-standing debate[8], concerns short-term memory, whose capacity is limited, and long-term memory, whose capacity is unlimited. According to a celebrated article (Miller, 1956), the span of short-term memory is restricted to seven items, plus or minus two, which restricts our ability to handle information. This number represents the number of isolated elements (numbers or letters not forming part of a meaningful sequence) that can be kept in short-term memory. This result has been used in computer science to develop complexity metrics. However, a complementary result, just as important, concerns the process of 'chunking'. This process extends the capacity of short-term memory by conceptually regrouping elementary items into items of a different sort. We shall see that, while the concept of short-term memory has been used effectively in the development of program complexity metrics, the equally important concept of chunking has not been taken into account.

Computer science has been interested in developing complexity metrics that allow subjects' performance in programming tasks, especially those that require the understanding of an existing program, to be predicted. Metrics were thus defined that were supposed to allow the comprehensibility of a program to be predicted[9]. McCabe, for example, defined a metric based on the control graph of a program. This metric represents the number of branches in the program.

Halstead took up the idea of a short-term memory span of seven units. He hypothesized that in a given task the items that a programmer would handle correspond to the operators and operands of the programming language. Following on from this, Halstead supposed that the cognitive effort used to understand a program ought to depend on a logarithmic function of the numbers of operands and operators in the program. He thus developed a static complexity metric involving these numbers to predict the comprehensibility of a program.

These metrics have been extensively criticised. As Curtis remarks[10], it is hardly credible to suppose that programmers, with even a little experience, handle information at a level corresponding to items as small as the operators and operands of a program. They must make use of meaningful units corresponding to what is expressed by statements or groups of statements. Through their domain experience, programmers must be able to build larger and larger chunks based on solution patterns. For example, the following might constitute a chunk corresponding to the concept 'calculate the sum of the elements of a table':

```
sum := 0
for i := 1 to n do sum := sum + table(i)
```

An experienced programmer would thus represent this sequence of instructions in the form of a single entity rather than as four unique operators (a total of seven operators) and six unique operands (nine operands in total).

1.1.5 General Criticisms

Finally, we observe that these early studies were severely criticized[11] and that the criticisms gave rise around 1980 to methodological and theoretical debates, which anticipated the theoretical and paradigmatic changes that were to be introduced later.

Hoc emphasizes that most of the studies avoid a psychological analysis of the activity of programming by using a superficial analysis of the task from a purely programming point of view. He condemns the systematic recourse to statistical inference without reference to a scientific psychological theory.

Moher and Schneider criticize the experimental methods, for example, in the choice of subjects (often novices) and the size of the (artificial) programs and tasks used in the experiments. This raises problems over the generality and applicability of the results. One may wonder whether the results obtained with novices in artificial and very constrained situations can be generalized to real situations. They wonder, indeed, whether these studies are, in fact, relevant to real programming. We shall see that the methodological and thematic evolution of later researches allows these obstacles to be overcome.

1.2 Second Period

From a theoretical point of view, the more recent period is characterized by the development of cognitive models of programming and, from a methodological point of view, by a change of paradigm. The use of clinical methods of activity analysis is advocated, alongside with, and as a complement to, strictly experimental methods. The research has two objectives. On the one hand, the theoretical objective is to enrich the theoretical frameworks borrowed from cognitive psychology. This enrichment is expected to come from the analysis of real-world tasks, such analyses being often lacking in experimental psychology. On the other hand, the practical objective is to improve the programming activity by adapting the software tool to its user. This ergonomic objective looks to increase the compatibility between the programmers' representations and the way they handle them, and the features of their working tools.

1.2.1 Theoretical Framework

Later research has been conducted mainly by psychologists and ergonomists or by multidisciplinary teams including psychologists and computer scientists. This research forms part of cognitive psychology. In this field, human activities are modelled in terms of representations and processing. The human being is thus considered as an information processing system[12]. The three most important concepts are representation, processing, and knowledge. Richard states

> What characterizes mental activities is that they construct representations and operate on them. The representations ... are essentially interpretations that consist in using knowledge to attribute an overall meaning to the elements that come out of a perceptual analysis, all this in the context of a situation and a particular task[13].

An important distinction is drawn between representation and knowledge:

> Representations are circumstantial constructions made in a particular context and for specific ends ... The construction of the representation is settled by the task and the decisions to be taken ... Knowledge also consists of constructions but they have a permanence and are not entirely dependent on the task to be carried out; they are stored in long-term memory and, as long as they haven't been changed, they are supposed to maintain the same form[14].

Representation is the cognitive content on which the processing takes place; a new representation is the result of the processing and can thus become the object of further processing. Representations are transitory. In effect, processes are linked to tasks and a new task gives rise to new representations. Some representations can enter the long-term memory, however, and some processes can be stored in the form of procedures.

This research is strongly oriented towards cognitive ergonomics inasmuch as it is concerned with real tasks, professional in the majority of cases. An important distinction is drawn between the concepts of task and activity. The activity is determined by a task. The latter is defined as the set of objective conditions that the subject is capable of taking into account in bringing into action his activity and the cognitive processes that underlie it[15]. It is a question of the objective elements of the situation concerning the end to be reached, the means available for reaching it, and the constraints affecting the deployment of these means. There is a fine distinction[16] to be drawn here between the *prescribed* task in a work situation, which defines what is expected of the subject, and the *effective* task which refers to the representation of the task that he or she constructs. The latter can be defined as the goal and the effective conditions taken into account by the subject.

The tasks that a programmer can carry out on a program already written are varied: detection and correction of errors (debugging), program modification, testing, reuse, documentation. The behaviour employed to reach a defined objective by a particular task is characterized by the activity that is observable and by the cognitive processes that underlie it.

1.2.2 Theoretical Changes

In the psychology of programming we thus pass from a period without theory to a period in which researchers borrow the theoretical frameworks coming out of cognitive psychology. Several pioneering articles thus began to develop such frameworks and also, but to a lesser extent, those from artificial intelligence:

categorization: the approach of Rosch and his colleagues[17] is applied to the categorization of programming problems[18];

understanding of natural language texts: the theory of Kintsch and van Dijk (1978) is applied to the understanding of programs[19];

learning: cognitive and educational models are borrowed to give an account of learning to program[20];

modelling of knowledge: schema theory[21] is borrowed by Soloway and his team[22] to account for the organization of experts' knowledge, and the theory of information processing on knowledge organisation in complex domains such as chess[23] is applied to the programming domain[24];

problem solving: Newell and Simon's theory of information processing (1972) is used by Brooks (1977) to take into account the cognitive mechanisms used in program design.

1.2.3 Paradigmatic Changes

As we have already explained, on the methodological plane, there are fewer isolated experimental studies which, while trying to quantify the effect of external factors on programming activity, ignore the cognitive mechanisms responsible for these effects. Recent researches are more of the clinical type, with subtle analyses of the activity according to the paradigm of verbal protocols. This has allowed descriptive models of the activity to be constructed.

This methodological development reflects, in fact, the evolution that has taken place in cognitive psychology during the same period. Alongside the experimental researches, sophisticated methods of observation have been developed, such as the analysis of individual protocols. The validation of hypotheses uses simulation methods at the same time as more refined statistical methods for hypothesis testing[25].

Mental activities can be inferred from behaviour and verbalisation and can be simulated by information processing models. In particular, the analysis of verbal and behavioural protocols allows the representations and cognitive processes deployed in an activity to be inferred. Verbalization is an expression of the activity where the subjects' representations and the rules linked to this activity are expressed clearly. Several authors have analysed and criticised the technique of verbalization used in sorting and problem-solving tasks[26].

1.3 Recent Thematic Developments

Research into the psychology of programming has undergone a major change in the last 15 years or so. In 1986 two papers[27], presented at the first workshop on Empirical Studies of Programmers, remarked upon the thematic directions and limitations of the researches carried out up to then in this field: many studies of tasks close to the activity of programming in its strict sense (coding), many studies of programming novices. These remarks raised the question of how far knowledge acquired in this field could be generalized.

The results of the studies were probably dependent on the lack of expertise of the participants, but to what extent? Further, the experimental situations studied, adapted for students, were certainly not representative of real software development situations, in several ways:

- the small size and low complexity of the problems used;
- the straightforward nature of the programming environment used;
- the purely individual character of the activity studied.

To use only novices as subjects did not therefore allow any real ergonomic recommendations for the development of complex programming environments to be made. Another limit to the generalization of the results was that the languages studied were mostly procedural or declarative and that the effect of the programming paradigm on the activity had not been studied.

The recent change addresses these worries through the emergence of research on the following themes:

- the activity of professional programmers and no longer just students;
- the collective and collaborative aspects of software development;
- the activities upstream of coding, e.g. specification and design;
- the effect of languages and programming paradigms on programming activities.

Certain recent researches have been centred on the activity of software experts, especially through the participation of professionals in experimental laboratory studies[28] or field studies[29]. These studies have allowed problems of realistic size to be studied, requiring sometimes several weeks or even several months of development. More generally, they have allowed the modelling and reasoning of experts in a complex field to be addressed.

Studying professional programmers, and not just beginners and students, led to the emergence of another research theme, the learning of new programming languages by experienced programmers. Previously, the learning theme had been restricted to the learning of a first language by programming novices. This new theme allowed the mechanisms of knowledge transfer between languages and even between paradigms to be addressed[30].

For a long time studies of programming centred on activities involving the manipulation of code, for example, code production, debugging, and the understanding of programs. More recently, activities more removed from the production or understanding of code have been studied. These concern the stages before the production of code, such as specification and design. Such research allows us to address the reasoning mechanisms used in solving design problems. Through the study of design, the theme of software reuse has also appeared.

The effect of programming paradigms is a recurrent theme in the literature but it is only very recently that their impact has been faced up to, notably through comparative inter-paradigm studies[31] and through the study of design using an object-oriented programming paradigm and not just a procedural paradigm[32].

Another recent thematic development is to consider programming not simply as an individual activity but also as a collective activity. At the level of a software development team, the themes addressed are the processes of co-operation among designers, the co-ordination processes, and the organizational aspects[33].

These recent studies highlight certain causes of difficulty that programmers experience in carrying out their work. The approach currently followed in developing software, in which, despite the emphasis placed on identifying user needs, there are no real empirical or ergonomic studies of user activity, is partly responsible for these difficulties.

From a practical point of view, research into the psychology of programming poses the problem of the distance between the representations and human processing, on the one hand, and the formal systems allowing these representations themselves to be represented and manipulated at a second level, on the other. From this point of view, the study of programming activities allows us to build models of these activities that can guide the development of programming languages, methods, and problem-solving aids for both expert and novice. This explains its interest for software engineering and will be a constant theme throughout this book.

References

1. See, for example, Curtis, 1982.
2. In particular, Sime, Green and Guest, 1973, 1977
3. Ibid.
4. Gannon, 1976.
5. Soloway and Ehrlich, 1984.
6. Wright, 1977.
7. Shneidermann, Mayer, McKay and Heller, 1977.
8. Florès, 1970.
9. Halstead, 1977; McCabe, 1976; Schroeder, 1983; Sunohara, Takano, Uehara and Ohkawa, 1981.
10. Curtis, 1980; Curtis, Forman, Brooks, Soloway and Ehrlich, 1984.
11. Brooks, 1980; Curtis, 1984; Hoc, 1982a; 1982b; Laughery and Lauchery, 1985; Moher and Schneider, 1982; Sheil, 1981.
12. Newell and Simon, 1972.
13. Richard, 1990, p. 9.
14. Ibid. p. 10.
15. Leplat and Pailhous, 1977.
16. Leplat and Hoc, 1983.
17. Rosch, Mervis, Gray, Johnson and Boyes-Braem, 1976; Rosch, 1978.
18. Adelson, 1981.
19. Atwood and Ramsey, 1978.
20. Mayer, 1981.
21. Schank and Abelson, 1977.
22. Soloway and Ehrlich, 1984.
23. Chase and Simon, 1973.

24. McKeithen, Reitman, Reuter and Hirtle, 1981.
25. Richard, Bonnet & Ghiglione, 1990.
26. Ericsson and Simon, 1980; Hoc, 1984a; Hoc and Leplat, 1983.
27. Curtis, 1986; Soloway, 1986.
28. See, for example, Détienne, 1990a.
29. See, for example, Visser, 1987.
30. Chatel, Détienne and Borne, 1992; Scholtz and Wiedenbeck, 1990a, 1990b; Wu and Anderson, 1991.
31. See, for example, Lee and Pennington, 1994; Petre, 1990.
32. See, for example, Détienne and Rist, 1995.
33. Bürkle, Gryczan and Züllighoven, 1995; Curtis and Walz, 1990; D'Astous, Détienne, Robillard and Visser, 1997, 1998; Herbsleb, Klein, Olson, Brunner, Olson and Harding, 1995; Krasner, Curtis and Iscoe, 1987; Robillard, D'Astous, Détienne and Visser, 1998.

http://www.springer.com/978-1-85233-253-2

Software Design – Cognitive Aspect
Detienne, F.
2002, XIV, 146 p. 10 illus., Softcover
ISBN: 978-1-85233-253-2