Chapter 3.3

Expert Software Design Strategies

Willemien Visser[1] and Jean-Michel Hoc[2]

[1]*INRIA – Ergonomics Psychology Group, Rocquencourt BP 105, F-78153 Le Chesnay, France*
[2]*CNRS – Université de Paris 8, URA 1297: Psychologie Cognitive du Traitement de l'Information Symbolique, 2, Rue de la Liberté, F-93526 Saint-Denis, Cedex 2, France*

Abstract

Early studies on programming have neglected design strategies actually implemented by expert programmers. Recent studies observing designers in real(istic) situations show these strategies to be deviating from the top-down and breadth-first prescriptive model, and leading to an opportunistically organized design activity. The main components of these strategies are presented here. Consequences are drawn from the results for the specification of design support tools, as well as for programmers' training.

1 Introduction

Although the top-down and breadth-first design model is very elegant, and rather simple to describe and to understand, professional programmers are aware of the fact

Psychology of Programming
ISBN 0-12-350772-3

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

that it is difficult to implement it in real software design. When teachers of computer science prescribe this kind of design strategy, they know very well that they themselves cannot – and do not – implement it in their design practice. Teaching complex domain knowledge to novices surely requires simplifying the concepts and the procedures. Simple ideas can guide novices in developing more complex skills. We have to tell novices, however, that the difficulties they encounter in implementing these simple ideas are not only due to their lack of knowledge, but also inherently linked to the simplistic nature of these ideas. In addition, it would be a mistake to refuse the complexity of expert design strategies when developing design tools that, otherwise, would only be useful in the simple cases for which elegant strategies are implementable. Moreover, seriously taking into consideration the difficulties programmers have when implementing these 'optimal' strategies could orient tool designers towards defining tools that could support more efficient strategies.

This approach then requires the identification of the strategies implemented in real design activity. Relatively few studies in cognitive science handle the question of software design by experts. Most research in the domain of programming concerns other activities and other subjects: the activities examined are, in general, conducted on already existing programs (mostly comprehension and memorization, and to a lesser degree debugging) and the subjects studied are mostly novices (see Chapter 1.3). Of course these studies can make contributions to design models:

* Comprehension studies, on the one hand, show which representations (often formalized in terms of 'schemas', see Chapter 3.1) the programmer possesses. These structures are certainly used in design and they constrain the strategies which are implemented, but specific design studies have to show *how* they are used, and *under which conditions*. On the other hand, these studies may contribute to the understanding of comprehension activities involved in design, that is, the comprehension of problem specifications, of other designs, and of modules already written in the design in progress.

* Studies comparing novices to experts show differences that may be characteristic of expertise. Their results constrain the definition of objectives for learning programs and teaching methodologies, and the type of tools and assistance to be developed.

1.1 Design: resolution of ill-defined problems

The most common conception of design problems is considering them as 'ill-structured' ones, in contrast to 'well-structured' problems which the psychology of problem solving has almost exclusively studied until now (Eastman, 1969; Simon, 1973).

Often a task that constitutes a problem for the person who is solving it – that is, for which this person has a representation that cannot trigger a ready-made procedure to reach the goal (Hoc, 1988b) – is considered as 'ill defined' by this problem solver. Following the problem space formalism, the main feature of a design problem concerns its solution, that is, the goal to be reached (Hoc, 1988b). This goal (a program in software design, a blueprint in architectural design, etc.) is objectively ill-defined; if not (a program or a blueprint is available), the problem is solved and no design activity is any more required.

To reach a suitable goal representation, the designer uses and transforms intermediary representations which constitute rough, underspecified, or even inappropriate, approximations of the goal:

* generally, initial problem specifications are not sufficient to define the goal, and stepwise definition of new constraints is necessary;

* the resolution of conflicting constraints, often existing between different levels, plays an important role;

* specifications or constraints come from different representation and processing systems (see Chapter 1.3), are often conflicting and have to be translated into a specific design domain – in software design, this domain corresponds to the programming language used;

* there is no 'definite criterion for testing any proposed solution' (Simon, 1973, p. 183), such as there typically exists for 'well-structured' problems: design problem solutions are more or less 'acceptable' or 'satisfying', they are not either 'correct' or 'incorrect';

* various different solutions are acceptable, one being possibly more satisfying in one dimension, another in another dimension.

Software design, which concerns us in this chapter, exhibits these characteristics. For example :

* the specifications given at the start are never complete or without ambiguity (that is, in real work situations);

* different programs, implementing different algorithms, may do for solving one and the same problem, even if one program may be judged 'better' than another, often for reasons of execution or maintenance, that is, for machine- or user-oriented efficiency criteria.

According to the type of software design, certain characteristics are more or less strongly present: text-processing software is surely less constrained at the outset than a programmable controller program that is going to govern an automated process.

1.2 Early design studies: sticking to the prescriptions

At the beginning, a trend seen in design studies, but not only there, was to 'conflate prescriptive and descriptive remarks' on the activity, and, rather than to consider what the activity *is really like*, to focus on what it *should be* (Carroll and Rosson, 1985). This led authors especially to describe design as well structured and even as hierarchically organized. This same tendency is observed in most, but especially the early, software design studies, such as the one by Jeffries *et al.* (1981). In the introduction to their article, the authors describe existing design methodologies, arguing that:

* these methodologies are 'indicative of the guidelines that experts in the field propose to structure the task';

* 'a reasonable model of performance ... ought to be related to accepted standards of good practice';

* 'most expert designers are familiar with this literature and may incorporate facets of these methodologies into their designs' (p. 256).

As a matter of fact, these methodologies advocate (modular) decomposition and provide different bases for performing it. As will be seen later, however, studies on professional programmers show that the theoretically optimal methods whose use has been learned and recommended need, to be implemented in fact, very particular conditions, whose realization is often not assured.

2 Software design studies

2.1 Methodology used

2.1.1 Protocol studies

In most of the empirical studies focusing on software design, the data come from protocol analysis. A limited number of subjects (from one to ten) is asked to think aloud during their activity. Generally, the verbal protocols of only a subgroup of these subjects are analysed in detail.

Brooks (1977) was the first author to construct, from the protocols collected on one programmer writing a large number of short programs, a general model of the cognitive processes in computer programming.

2.1.2 More or less simple problem statements

Most of these studies (especially Jeffries *et al.*, 1981, and Adelson *et al.*, 1985) use experimenter-constructed, rather artificially limited problem statements.

Hoc (1988a) uses problem statements that lead to programs of small size, but which control algorithmic difficulties. These problem statements have been constructed in order to represent the categories of a typology of programming problems, resulting from a previous empirical study on problem classification in relation to the type of problem-solving strategy.

Guindon *et al.* (1987) judge the problem they use to be 'more complex and realistic ... than [those that have] been given in other studies of software design, yet not so different that [their] results cannot be easily compared to them' (p. 66). The problem they use is, however, incomparable, in complexity and degree of realism, with real design problems such as studied by Visser (1987).

Visser (1987) conducted a rather different, up to now – as far as we know – rather unique, study. Full-time observations were made on a professional programmer, in his real working environment (a machine tool factory), for a period of four weeks. The programmer was solving a real, complex industrial problem (control of a machine tool installation). Visser observed the programmer's normal daily activities without intervening in any way, other than by asking him to verbalize as much as possible his thoughts about what he was doing. In her data analysis, she focused on the specificity of the strategies used in real work conditions. Using the same method, observations were conducted on the mechanical engineer, during his design of the (functional) specifications for the programmer (see Visser, 1988a,b, 1990).

This methodological choice of observations on large software design projects is necessary. More often than not, observations or experiments are done on small software design problems and selection of problem statements is rather anecdotal. This leads to questioning the ecological validity of the results. Nevertheless, the size of the program is not the only relevant dimension for evaluating design difficulty. A large program, if it is familiar to the programmer, can take a long time to write, without any actual problem-solving activity, but requiring only coding. Conversely, a small program can accurately represent some problem-solving processes that are used in the development of large projects. As observations on small programs are more easily conducted on several programmers, they especially enable the evaluation of individual differences. Certain features of large software design, however, are impossible to reproduce when one reduces the program size (for example, the working memory management).

2.2 Models constructed

From his protocol analysis, Brooks (1977) elaborates a programming model, in which he distinguishes, next to the *coding* on which the model focuses, two other types of activities:

* *understanding*, leading to a problem representation;
* *method finding*, that is, construction of a plan, a (hierarchically) organized program representation (using relations between goals and subgoals) – an important role is occupied here by two kinds of knowledge structures, computer science and task domain-related schemas.

Adelson *et al.* (1985) propose a general model of the design activity, functioning with four components:

* a 'design meta-script', that is, a high-level schematic representation whose function is to drive the design process by setting goals for processing the 'sketchy model';
* the 'sketchy model', that is, the current solution state, which becomes progressively less sketchy, that is, more concrete and elaborate, until the implementation level representation;
* the 'current long-term memory set', consisting of all the known solutions appropriate to the aspect of the design that is currently being worked on;
* the 'demons', which monitor the state of the 'sketchy model', activating 'things to remember', provide elements to elaborate and modify the Sketchy Model into a final, concrete design solution.

Other authors stress the structures controlling the activity (Jeffries *et al.*, 1981; Guindon and Curtis, 1988).

Brooks (1977) and Jeffries *et al.* (1981) formulate production rules to account for the activity covered by the protocols. Adelson *et al.* (1985) use goals and operators organized in a goal hierarchy. Guindon and Curtis (1988) only describe the components of the model and their articulation.

3 Different strategies used in designing software

3.1 Variability between experts

Most authors note *variations in the design strategies* and the *solutions* they lead to, not only between levels of expertise when they compare experts and novices, but also between experts. Rather than considering this variability as a nuisance factor or a marginal result, we judge it as inherent and characteristic to design, as described in the introductory section, that is, the development of *a*, not *the*, solution to a problem that is not completely defined at the start. In such conditions, different designers will proceed in different manners, introducing different solution elements at different moments (see also Falzon and Visser, 1989, who show that, due to different past task experiences, different expert designers may exhibit different *types* of expertise).

3.2 Global control strategy: problem decomposition

Almost all subjects in early design studies are observed to use the same global control strategy advocated by design methodologies, that is, *decomposition* of the problem into subproblems. But several decompositions are possible for a problem, bearing on rather different principles.

In a study on more or less advanced computer science students, Ratcliff and Siddiqi (1985) identify two types of problem decomposition:

* data driven – the generation of the program structure is guided by a mental execution strategy, which bears on a simple representation of the input data, just sufficient to satisfy processing requirements;

* goal driven – the analysis of the goal structure leads to a non-trivial representation of the input data, which is more declarative than procedural, resulting in a quite different problem decomposition.

Two remarks may be formulated concerning the general nature of decomposition. Firstly, the use of this strategy may depend on expertise. The very beginner observed by Jeffries *et al.* (1981) does not succeed at all in decomposing problems, whereas their advanced students generate decompositions rather in terms of successive processing steps, than in terms of modules.

Secondly, as shown by Guindon *et al.* (1987), there are partial design solutions that are not the result of a decomposition, but which are, for example, retrieved 'by recognition'. As a consequence, such solution elements do not always fit into a balanced global solution.

If Jeffries *et al.* (1981), and Adelson *et al.* (1985) even more, stress the 'neat' predictable structure of decomposition strategies, and in general of the problem-solving activities involved in design, Guindon *et al.* (1987) and Visser (1988a,b) insist on the deviations of these structures. Moreover, the 'breakdowns' (Guindon and Curtis, 1988) observed do not always reflect deviations from predictable decomposition structures: many of them are rather caused by the opportunistic character of design.

3.3 Top-down and bottom-up strategies

The top-down strategy consists in descending the solution tree from the most abstract level down to the lowest, concrete level, never coming back up to a higher level (*top-down refinement*).

Based on his observations, Brooks (1977) expected that, only if a programmer is working on a problem with which he is very familiar and if he resolves it in a programming language in which he has considerable experience, he may proceed sequentially, without backtracking, through the three stages identified by the author (understanding, method-finding and coding). That is, only expert programmers, and even they only in these particular conditions, may proceed in a strictly top-down fashion.

Most design studies confirm this prediction.

For example, professional programmers working with a computer support to top-down processing that renders plan revision tedious generate non-optimal solutions: planning errors are sometimes rescued by awkwardly modifying modules at a very low level (Hoc, 1988a).

In the study of Jeffries *et al.* (1981), only one expert (out of four) showed a systematic implementation of a top-down strategy. The other experts deviated more or less from this 'optimal' strategy. The authors note that a designer may choose to deviate from the advocated order when he realizes that a component has a known solution, is critical for success, or presents special difficulties. This is typically one of the ways of proceeding qualified by Visser (1987) as 'opportunistic' (Section 3.5).

3.4 Breadth-first and depth-first strategies

Design methodologies describe and advocate the use of a breadth-first strategy – combined with a top-down strategy – for decomposition: when decomposing the current level solution, one should develop all the elements of the new solution at the same level of the solution 'tree' and integrate them into a new global structure, rather than refining, until its final solution elements, one or several particular solution branches (which would be *depth-first* processing).

All three experts observed by Adelson *et al.* (1985) implemented this strategy (called '*balanced development*' by the authors).

The one expert (out of four) in the study of Jeffries *et al.* (1981) observed to proceed systematically top down, did so in combination with a breadth-first strategy. Other experts were observed to follow a rather depth-first strategy, starting, for example, their decomposition by a top-down processing of only some branches of the tree, handling the other ones afterwards.

For handling interaction, breadth-first processing is of course very useful, even if the detection of potential interactions may require descending branches in anticipation.

Handling a problem at one level, one may think of related elements at another level. Sometimes, experts are capable of maintaining these kinds of elements in memory and retrieving them at the appropriate moment. Adelson *et al.* (1985) observe their experts making 'notes to themselves' (concerning constraints, partial solutions or potential inconsistencies) and the authors posit the existence of 'demons' reminding the designer to incorporate this information into the design once the appropriate level of analysis has been reached.

Hoc (1988a) and Guindon *et al.* (1987) observe, however, that even experts have difficulties considering and maintaining simultaneously all problem or possible solution elements at one level of abstraction. They observed subjects engaged in bottom-up processing activities and noticed the difficulties these subjects encountered, such as backtracking of subproblems whose solution had been postponed or whose solution had to be modified. In the situation analysed, Hoc ascribes these difficulties to the nature of the environment. Following a top-down strategy, the language and the editor used in this experiment constrain the subjects to express too precise expressions in the design. For example, when one is considering an iterative structure for a module at a high level in the tree, a precise representation of the adequate control structure can be unavailable. If the environment imposes the choice too early, the users have to analyse the problem at the next level. Then they write the result of this analysis in order to preserve their memory load and adopt a depth-first strategy.

In a study on expert programmers, Petre and Winder (1988) confirm the need for languages that can support different levels of analysis in the course of design. They notice that, before introducing the constraints of actual programming languages, experts very often use a personal pseudo-code.

3.5 Opportunistic strategies

As noticed above (see Section 3.3), only one expert (out of four) in the study of Jeffries *et al.* (1981) implements a 'pure' top-down strategy. The other three have been observed to display the following behaviours:

* starting the decomposition in the middle of the tree;
* working simultaneously on two distinct branches;
* making interruptions for digressions at other than the current level, for example, to deal with other subproblems or to define primitive operations, that is, elements at the lowest level;
* descending in the decomposition tree, but coming back up afterwards, for example to introduce a whole new solution decomposition level.

Guindon *et al.* (1987) noticed, at least, two types of returns: on the one hand, to tentative solutions proposed earlier. On the other hand, they observed that subjects, in an advanced design phase, re-examined the specifications to understand them. The designers studied by Visser (1988a,b) in their real working environment also proceeded to such re-examinations; however, they modified not only the specifications they received, but even those having governed anterior design stages and thus underlying their specifications.

At the onset of his design, the professional programmer observed by Visser (1987) decomposed the problem into functional modules which he planned to handle in a top-down way. Afterwards he often deviated from this plan, organizing his activity rather opportunistically (see Hayes-Roth and Hayes-Roth, 1979). Two important factors determining the guidance of his problem solving – causing the activity to be opportunistic – were the cognitive cost of actions and their importance (see Visser, 1990). Examples illustrating each one of these factors are the following:

On the one hand, if information required for handling the current design component was not, not yet, or not easily available, its processing was often *postponed* – because it would have been 'expensive' – leaving unfinished modules at the current level of the design. On the other hand, information was sometimes processed only because its processing was 'cheaper' than proceeding to the according-to-the-plan action. An example is the processing of the information provided by the information source at hand, rather than looking for the information required to handle the current design component. This then might lead to the programmer defining modules *in anticipation* and/or *at other levels of detail and abstraction* than the current one.

An action may be important because of the type of action or because of the object concerned by the action. For example, 'verifying' is an important action only if the verification concerns certain objects.

3.6 Prospective and retrospective, procedural and declarative strategies

The two types of problem decomposition identified by Ratcliff and Siddiqi (1985) were explained with reference to two design strategies: data driven or goal driven. After a study of problem classification by professional programmers in relation to design strategies, Hoc (1988b) proposed a more complex framework to classify problems and strategies from this point of view.

Data-driven strategies, as observed by Ratcliff and Siddiqi (1985), are of a *procedural* kind: the program is generated following a mental execution strategy. The statements are written in the order of execution and the writing is guided by an available procedure. This is a *prospective* procedural strategy very often encountered when the problem statements trigger a quite familiar procedure (for instance, execution by hand).

But a prospective strategy can be more *declarative*. This is the case, for example, in management problems where the complex structure of the input files and the relationships between them introduce strong and complex constraints on the program structure. Then a static representation guides the design. In other problems, the guide can be provided by the structure of the output files and their relationships: the program can be written following the reverse order and the strategy is *retrospective* declarative. These declarative strategies are quite well supported by diverse management programming methodologies (for example, Jackson or Warnier methods).

A retrospective strategy can be more procedural than declarative when guidance is given rather by goal-subgoal or preconditions relationships than by a static output file structure. This retrospective procedural strategy is often implemented by novices, as has been observed in physics problem solving (Larkin and Reif, 1979). The novices start from unknowns, they search for definitions of these variables in their data base, generate new unknowns (intermediary results), etc., until reaching given values. On the contrary, experts can classify problems in categories for which they have available procedures: they may then develop prospective procedural strategies.

In the experiment cited, Hoc (1988a) used this classification of problems and strategies to assess a programming environment especially designed to help professional programmers developing a structured, top-down and retrospective programming method. Although prospective strategies were not hindered by the environment, the experiment shows that retrospective strategies were strongly induced by the environment (whatever the type of problem). Prospective problems were solved with greater difficulties than retrospective ones. Nevertheless, retrospective problems were not so easy to solve in the environment: difficulties appeared to be explained by the lack of appropriate data structures in the language. When following a retrospective strategy, guidance cannot be given by the mental execution of the procedure and must be provided by data structures and relationships between them.

Results of this kind show that sticking to a too rigid design methodology, supposed to be valid whatever the type of problem (or subproblem), is to be avoided. The domain of validity of a methodology should rather explicitly be defined in terms of a typology of problems.

3.7 Simulation

Adelson *et al.* (1985) stress the frequent occurrence of this strategy and its importance for the experts they observed. The mental models their subjects constructed were run as they were elaborated at different levels of abstraction. These simulation runs are supposed to assist the designer on, at least, two points:

* predicting potential interactions between elements of the design;
* pointing out elements of the solution state that need expansion.

Maintaining the balanced development is considered by the authors to serve these simulations, which require all elements of the model that is run to be at the same level of detail.

Simulation may serve different objectives:

* comprehension, when the designer explores and simulates the problem environment;
* evaluation, when he runs simulations of tentative solutions, and selects between them, for example, on criteria of efficiency.

Guindon *et al.* (1987) observed that these 'exploratory' design strategies are used for (sub)problems for which specialized design schemas cannot be evoked. They insist especially on the high frequency of occurrence of the simulation-for-comprehension strategy in their study compared to the results obtained in other design studies.

The programmer observed by Visser (1987) used simulation of the program's execution, mainly to understand the specifications. Simulation of the installation's operation, generally considered to characterize novices, was sometimes used by this experienced programmer to check modules of the part of the program he had already written. These simulations were among the rare moments the programmer verbalized his thoughts spontaneously.

3.8 Reduction of complexity

Considering a problem under its most typical form, modifying it only later to take into account its specific conditions, is a cognitive economical strategy. It requires, however, that the designer has at his disposal categories of problems and associated solutions ('schemas'), differentiating these different elements by their appropriate attributes. So it seems to be reserved to expert designers.

Various authors observed this kind of strategy. So did Hoc (1988a), but he noticed the environment precluding the implementation of such a generalization strategy. Indeed, this approach made it difficult to modify an initially developed solution with the editor used in his experiment.

The designers observed by Guindon *et al.* (1987) used another type of complexity reduction strategy, when they generate simplifying assumptions which they afterwards evaluate for their plausibility.

3.9 Considering users of the system to be designed

Both Adelson *et al.* (1985) and Visser (1987) observe their subjects to be guided, more or less, by such considerations, but with different functions in the two studies.

In the first one, mental *simulation* of a user's interaction with the system helped the designer to *think of elements to be included* in the design.

For the programmer observed by Visser, the ease of use for future users (system operators as well as maintenance personnel) was a *criterion of evaluation* of his design: considering homogeneity an important factor of ease of use, this led him to *make the program as homogeneous as possible.*

One may suppose that this strategic consideration has not been observed to be implemented in more studies, because of its link to – and perhaps dependency on – real work situations. Adelson *et al.* (1985) noted it, however, even if – contrary to Visser (1987) – they made their observations in a restricted laboratory setting.

3.10 Use of past experience and other knowledge

The use of knowledge has been examined much more in program comprehension studies (see Chapter 3.1) than in the design studies presented here.

3.10.1 Use of software design knowledge

Solutions, next to being constructed in an actual problem-solving activity, may be arrived at by retrieval – and of course adaptation – of a stored solution, which may be modified or not. The retrieved solution may come:

* from memory – for example, algorithms learned by using them in the past, or a published algorithm that has been retained (Jeffries *et al.*, 1981);

* from an external source – for example, an earlier written program (Jeffries *et al.*, 1981).

Visser (1987) observed the programmer relying heavily on existing (partial) solution instantiations (that is, listings of programs written in the past and parts of the program-in-progress). Once again, one may think that most psychological studies paying so little attention to this reuse – contrary to software engineering, which considers it to be a major problem to be solved – is due to their rather artificially limited context making reuse difficult to implement: to reuse a design module in the resolution of a problem, this problem must be similar to those the designer has processed in the past.

3.10.2 Use of problem domain knowledge

Knowledge about exemplars of the kind of system to be designed or about functional requirements of systems in general may be used to constrain the definition of the design-in-progress or to retrieve solution elements (Adelson *et al.*, 1985).

Visser (1987) noted that schema-guided information processing could explain certain errors made by the programmer, for example, when he violated the specifications for an 'atypical' function, by defining it as a completely typical function. The programmer's expectations – based on prototypical schema slot values – were probably so strong that he did not take into account the values which were given (in the specification document) (see Détienne, 1990).

4 Assistance to the design activity

The results of the presented studies provide a rationale for the specification of assistance tools to support designers.

4.1 Displays for helping the management of working memory

One may think of tools enabling *parallel presentation of intra- or inter-level information* (see the difficulties on breadth-first decomposition) or presentation of all *constraints on the solution order* or *maintaining a trace of postponed subproblems* needing backtracking (see the 'Design Journal' suggested by Guindon and Curtis, 1988, which could have still other functions).

4.2 Libraries of design schemas

Visser (1987) noticed the importance of examples and of past designs reuse for the programmer she observed. As long as a designer wishes to use his own past productions, such libraries are not too difficult to realize. But the interest of this function lies especially in providing designers with the experience of colleagues. In this case, problems of indexing, for example, arise.

4.3 Assistance for the articulation of top-down and bottom-up components

As a purely top-down decomposition is rarely implemented, such assistance would be useful. In the environment evaluated by Hoc (1988a), the articulation between these two components was really precluded.

4.4 Assistance to prospective strategies

As Hoc (1988a) concludes, this kind of assistance is more difficult to implement than assistance to retrospective strategies (as provided in the environment he evaluated). Subjects' goals are indeed more difficult to infer than the prerequisites of explicitly enunciated goals.

Structure-based editors have been proposed in order to aid prospective strategies.

4.5 Assistance to simulation

Given the importance of simulation in progressively developed design, assistance to this function would also be useful. As simulation often involves holding simultaneously several variables in mind, one could think of supporting the management of this memory load – a function that would not be particular to, but especially important for, simulation assistance.

The development of all these tools requires more research into the processes and components of the strategies to be assisted.

5 Conclusion

The different studies presented in this chapter did not all come up with the same results. Especially between those of Jeffries *et al.* (1981) and Adelson *et al.* (1985), on one side, and those of Guindon *et al.* (1987) and Visser (1987), on the other side; an important difference concerns the systematic use of rather 'optimal' decomposition strategies, such as top-down and breadth-first processing, which the first authors noticed to characterize their designers. Guindon *et al.* (1987) and Visser (1987) seemed to observe many more deviations from these strategies, leading them to consider 'opportunism' as an important factor of design activity.

An explanation for this difference might be found in the type of design problem to be solved: real design problems (as used by Visser, 1987, and approximated by Guindon *et al.*, 1987) or restricted problems (as used by the others). Further studies of expert design activity on more or less realistic problems could confirm this hypothesis. Working on a computational geometry algorithm design, Kant (1985) notices that 'control [of design activity] ... comes out of responding to the data and out of the problems and opportunities arising during execution' (p. 1366). Ullman *et al.* (1987) conclude that 'mechanical designers progress from systematic to opportunistic behaviour as the design evolves' (p. 157).

However, the difference we notice may also be due (to some extent) to the perspective the authors take on their data. As mentioned in the introductory section, early design studies often stuck strongly to the normative viewpoint. Both studies concluding on the systematic nature of top-down and breadth-first strategies (Jeffries *et al.*,1981; Adelson *et al.*, 1985) are among the first studies conducted on expert software design. Psychologists know the role expectations play on the processing of information.

References

Adelson, B., Littman, D., Ehrlich, K., Black, J. and Soloway, E. (1985). Novice-expert differences in software design. *In* B. Shackel (Ed.), *Human-Computer Interaction – INTERACT 84*. Amsterdam: North-Holland.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies,* **9**, 737-751.

Carroll, J. M. and Rosson, M. B. (1985). Usability specifications as a tool in iterative development. *In* H. R. Hartson (Ed.), *Advances in Human-Computer Interaction,* vol. 1. Norwood, NJ: Ablex.

Détienne, F. (1990). Program understanding and knowledge organization: the influence of acquired schemata. *In* P. Falzon (Ed.), *Cognitive Ergonomics: Learning and Designing HCI.* London: Academic Press, pp. 245-256.

Eastman, C.M. (1969). Cognitive processes and ill-defined problems: a case study from design. *In* D. E. Walker and L. M. Norton (Eds), *Proceedings of the First Joint International Conference on Artificial Intelligence.* Bedford, MA: MITRE.

Falzon, P. and Visser, W. (1989). Variations in expertise: implications for the design of assistance systems. *In* G. Salvendy and M. Smith (Eds), *Designing and Using Human-Computer Interfaces and Knowledge Based Systems.* Amsterdam: Elsevier.

Guindon, R. and Curtis, B. (1988). Control of cognitive processes during software design: What tools are needed? *In* E. Soloway, D. Frye and S. S. Sheppard (Eds), *CHI'88 Conference Proceedings.* Reading, MA: Addison-Wesley.

Guindon, R., Krasner, H. and Curtis, B. (1987). Breakdowns and processes during the early activities of software design by professionals. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Hayes-Roth, B. and Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science,* **3**, 275-310.

Hoc, J. -M. (1988a). Towards effective computer aids to planning in computer programming. Theoretical concern and empirical evidence drawn from assessment of a prototype. *In* G. C. van der Veer, T. R. G. Green, J. M. Hoc and D. Murray (Eds), *Working with Computers: Theory Versus Outcomes.* London: Academic Press.

Hoc, J.M. (1988b). *Cognitive Psychology of Planning.* London: Academic Press.

Jeffries, R., Turner, A. A., Polson, P. G. and Atwood, M. E. (1981). The processes involved in designing software. *In* J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

Kant, E. (1985). Understanding and automating algorithm design. *IEEE Transactions on Software Engineering,* **11**, 1361-1374.

Larkin, J. H. and Reif, F. (1979). Understanding and teaching problem solving in physics. *European Journal of Science Education,* **1**, 191-203.

Petre, M. and Winder, R. (1988). Issues governing the suitability of programming languages for programming tasks. Paper presented at *ECCE4 - Fourth European Conference on Cognitive Ergonomics.* Cambridge, September 1988.

Ratcliff, B. and Siddiqi, J.I.A. (1985). An empirical investigation into problem decomposition strategies used in program design. *International Journal of Man-Machine Studies,* **22**, 77-90.

Simon, H. A. (1973). The structure of ill-structured problems. *Artificial Intelligence,* **4**, 181-201.

Ullman, D., Staufer, L. A. and Dietterich, T. G. (1987). Preliminary results of an experimental study of the mechanical design process. *Proceedings of the Workshop on the Study of the Design Process.* Oakland, CA, February 1987.

Visser, W. (1987). Strategies in programming programmable controllers: a field study on a professional programmer. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Visser, W. (1988a). Giving up a hierarchical plan in a design activity. Research Report No. 814. Rocquencourt: INRIA.

Visser, W. (1988b). Towards modelling the activity of design: an observational study on a specification stage. *Proceedings of the IFAC/IFIP/IEA/IFORS Conference Man-Machine Systems. Analysis, Design and Evaluation,* vol. 1. Oulu, Finland, June 1988.

Visser, W. (1990). More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies* (special issue on empirical studies of programmers), in press.