the complex, multi-dimensional character of the representation constructed. A program can therefore be represented as a tree of goals and sub-goals, with objects represented as central or secondary entities, and plans for attaining the goals. Different types of relations, such as data flows, communication between objects, the structure of plans, can be represented.

On the other hand, this definition does not restrict programming simply to the production and expression of a procedure in a programming language. Due to the special use for which computer programs are intended, execution by a machine and use (producing or reading) by a human, the final product includes at the same time instructions in the programming language and in natural language. We therefore consider that, at the level of its surface structure, a program is a 'text' written in a programming language and in natural language. We shall see in this book that the information expressed in each of these two types of language has its own role to play in design and understanding.

References

1. Kant and Newell, 1985; Pennington, 1987a; Pennington and Grabowski, 1990.
2. Rist, 1991.
3. Hoc, 1987a.
4. Mayer, 1987.
5. Robertson and Yu, 1990.
6. Baudet and Cordier, 1995.
7. Robertson, 1990.
8. Puglielli, 1990.
9. Bellamy, 1994a; Davies, 1996; Henry, Green, Gilmore and Davies, 1992.
10. Détienne, Rouet, Burkhardt and Deleuze-Dordron, 1996.

3. *Software Design: Theoretical Approaches*

A special feature of software design is that the problems to be solved are 'ill-defined'. There is also a distinction to be drawn between the problems of producing programs and the problems of producing results. We shall discuss these characteristics in Section 3.1. We then go on to present three theoretical approaches used in studying the activity of software design: the knowledge-centred approach, the strategy-centred approach and the organization-centred approach.

The knowledge-centred approach, discussed in Section 3.2, involves identifying and formalizing the knowledge of the expert within the framework of the theory of schemas. According to this approach, the activity of designing a program consists, in part, of activating schemas kept in memory that are suitable for handling certain problems, and of combining them to construct a program. This work concentrates on analysing the knowledge of experts and pays little attention to the way the knowledge is used. Thus it largely ignores the strategic aspect of program design.

The strategy-centred approach is discussed in Section 3.3. It seems that the expert is characterized, not only by more abstract knowledge, organized hierarchically, but also by a wider and more adaptable range of strategies than the novice. Design strategies can be described in terms of several different features, for example, whether they are top-down or bottom-up. An important research question is to specify the condition for adopting particular strategies. Externally the choice is determined by the characteristics of the notation, the features of the environment, and the problem type. The internal determinants are linked to the subject's expertise in computing, especially the availability in his or her memory of programming schemas. But it is often the interaction between several factors that determines the strategy.

Section 3.4 is concerned with the approach which is centred on the organization of the design activity, a more 'meta' level than the strategy-centred approach. Two types of model contrast with each other:

1. The hierarchical model based on normative models inspired by programming methods.
2. Opportunistic models, based on the results of empirical studies, which emphasize how the real activity deviates from a strictly hierarchical model.

Finally, some studies place the accent on the iterative character of the design activity, drawing a parallel with models of text production. We shall also look at the note-taking mechanisms that are used in design, demonstrating that documentation is an inherent part of the design process.

What distinguishes an expert from a novice? What are the formative stages of the expertise? Can we identify several different types of expertise? These questions will be discussed in Section 3.5.

Studies of program design have practical implications, especially for the ergonomic specifications of programming environments best adapted to their users. This is the topic of Section 3.6, while the final section explores the limits of existing work and the prospects for the future.

3.1 Features of the Problems of Program Design

3.1.1 Ill-defined Problems

Program design has been mainly studied within the framework of research into problem-solving activities. The problem-solving activity is usually described as being made up of three successive phases; understanding the problem, research and development of the solution, and coding the solutions. In fact, these phases are not purely sequential because there is interaction between them.

Program design, like other design activities (architectural design, for example) has the special characteristic of being ill defined, in the sense that some of the specifications of the problem are missing and part of solving the problem is to introduce new constraints. Another feature is that there exist several acceptable solutions for the same problem, which cannot be evaluated according to any single criterion.

An equally important aspect of the program design activity is that designers use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping. Simplifying, we can say that programmers construct at least two types of mental model; a model of the problem and its solution in terms of the application domain and a model in computing terms. Part of their work consists of passing from one model to the other. Depending on the features of the design situation, the distance between these models will be bigger or smaller. One might hypothesize that some programming paradigms, for example, object-orientation, reduce this distance for particular application domains.

3.1.2 Problems of Program Production

In general, solving a problem is the same thing as working out a solution. However, two different types of problem-solving situation[1] can be distinguished according to what type of solution is produced, either a procedure for obtaining a result or the result itself:

1. In a result-producing situation, the subject concentrates on the goal assigned and, if a procedure is developed, this is only secondary. It will usually be stored in memory in executable form. The subject has not therefore constructed a representation of this procedure.

2. When a program has to be produced, the subject concentrates on working out a procedure. This means constructing a representation of the structure of the procedure over and above the result required.

The distinction between the two situations is well illustrated by the task of sorting a set of names into alphabetical order. Most literate people can produce the required result with little difficulty. It is much harder, however, to describe a procedure for carrying out this task in general.

Clearly, designing a program is much more like the second kind of task. It consists, indeed, in constructing a representation of a procedure and, further, of expressing this procedure in a certain design notation or language.

3.2 Knowledge-centred Approaches

As we shall see, identifying and formalizing the knowledge of expert programmers has given rise to a great deal of research.

3.2.1 Theory of Schemas

Researchers in program design are generally agreed that there are three types of knowledge that serve to distinguish experts from novices:

1. Syntactic knowledge, which defines the syntactic and lexical elements of a programming language, for example, the fact that, in C, the **if** statement takes the form **if** (*condition*) *statement*.
2. Semantic knowledge, which refers to the concepts, such as the notion of a variable, that make it possible to understand what happens when a line of code is executed.
3. Schematic knowledge, that is, programming schemas that represent generic solutions.

The theory of schemas has been widely used to describe the knowledge of expert programmers[2]. It is a theory of the organization of knowledge in memory and of the processes for making use of this knowledge. A schema is a data structure which represents generic concepts stored in memory[a]. The notion of a schema was developed in artificial intelligence and in psychological studies of text understanding.

Before going further into schemas we must make a distinction between the idea of a schema and the idea of a solution plan. A schema is a knowledge structure. A solution plan is a sequence of actions in a program which will achieve the goal of the program. (This idea is illustrated in Fig. 2.3 of Chapter 2.) For the expert, a plan often represents a special case or an instance of a programming schema. A schema being a structure of variables to which is associated a set of possible values, the special case is characterized by a combination of specific values chosen to achieve a specific goal in a specific program.

[a]The concept of a *frame* is widely used in artificial intelligence and there seems to be no consistent difference in usage between the terms *frame* and *schema*.

3.2.2 Programming Schemas

Studies of programming show that expert programmers keep in their memory schemas specific to the programming domain. We can classify such schemas into the following two types:

- Elementary programming schemas representing knowledge about control structures and variables (called 'control plans' and 'variable plans' by Soloway). For example, a counter variable schema can be formalized as following:
  - Goal: count the occurrences of an action
  - Initialization: count:= n
  - Update: count:=count+increment
  - Type: integer
  - Context: loop

  Schemas of this sort can be thought of as consisting of a frame and slots, as this example illustrates. Prototypical values are zero for the initialization and one for the increment. Plans which are instances of elementary programming schema are also illustrated in Fig. 2.5 of Chapter 2;
- algorithmic schemas or complex programming schemas represent knowledge about structure of algorithms. For example, some programmers will be familiar with a variety of algorithms for sorting and searching. These algorithms are more or less abstract and more or less independent of the programming language, and they can be described as made up of elementary schemas. For example, a sequential search schema is less abstract than a search schema and can be described as being composed, in part, of an counter variable schema.

Another way of classifying programming schemas is according to their degree of dependence on the programming language[3]:

- tactical and strategic schemas, which are independent of the programming language (or, at least, within a single programming paradigm);
- implementation schemas, which are dependent on a particular programming language.

More strictly, we should refer to dependence on particular features of programming languages. Implementation schemas for operating on trees may be dependent on the possibility of recursion in the programming language; many but by no means all languages support this.

Schemas are related to each other in various ways:

- composition, that is, a schema is made up of several simpler schemas;
- specialization, i.e. the 'is a' relationship. A sequential search schema 'is a' search schema and a counter-controlled running total loop schema 'is a' running total loop plan;
- implementation, i.e., an implementation schema in a specific programming language implements a more abstract tactical or strategic schema that is language independent. Thus a for-loop schema in C or Pascal is an implementation of an abstract counter-controlled running total loop schema;

Rist (1986) introduced the notion of the 'focus' or 'focal line' of a schema. The focus is the part of the schema that directly implements its goal. Thus, for the counter schema, the focus would be the incrementing of the counter. The focus is the most important part of the schema. Certainly it is the part that is most available when a schema is retrieved. In design, the focus is the part that is recovered and expressed first, the code being constructed around it.

3.2.3 Other Types of Schema

Programming schemas, which are rich in knowledge and content, can be contrasted with structural schemas. In studies of the production and understanding of text, especially those using the story grammar approach, the structural schemas are often called the superstructure. A text of a certain type (e.g. a scientific article) can be described by a structure specific to that type of text. The knowledge of this structure, remembered in the form of the superstructure, can thus guide the production and understanding of this type of text.

The notion of role, introduced by Rist (1986) in the programming field, can be considered similar to this notion of structural schema. For example, a structural schema for a functional program is composed of three roles: input, calculate and output. A structural schema for classes of object-oriented programs is composed of the following roles: creation, initialization, reading access, writing access, input and output.

Programmers' ability to write or understand programs depends also on their familiarity with structural schemas.

Domain specific schemas, like programming schemas, are rich in knowledge and content. Détienne (1986) suggests that, as they become familiar with a problem domain, experts develop domain-specific schemas, that is, knowledge schemas representing their knowledge of certain types of problem. Thus an expert in invoicing and sales ledger applications will have a schema for discount structures.

3.2.4 Rules of Discourse

Experts also have rules of discourse that control the construction of programs and especially the instantiation of schemas during design. When designing a program, experts retrieve suitable programming schemas from memory and instantiate them according to the particular problem they are solving. This instantiation is controlled by the rules of discourse. Three typical examples[4] are:

- the name of a variable must reflect its function;
- if a condition is tested, the condition must have the potential of being true;
- do not use the same piece of code for two different purposes in a non-obvious way.

In professional software engineering organizations, the rules of discourse are usually formalized in sets of coding standards for the different programming languages that the organization uses. Checking programs to ensure that they comply with these standards is a normal part of the quality assurance procedures.

3.2.5 Limitations of Schemas

The theory of schemas allows us to take into account certain cognitive mechanisms used not only in programming design but also in learning programming and in understanding a program. Program design consists, in part, of activating schemas held in memory, suitable for handling certain problems, and in combining them to build a program. Learning to program is characterized by the progressive construction of programming schemas. Understanding a program consists, in part, of activating schemas stored in memory using indexes extracted from the program code and then inferring certain information starting from the schemas retrieved (cf. Section 6.3.1). This approach in terms of schemas is limited because it takes little account of other processes that are found in these activities, which are bottom-up and more constructive.

3.3 Strategy-centred Approach

All the work described above is centred on the analysis of the knowledge of experts rather than on analysis of the way this knowledge is put to use. In other words, it does not address the strategic aspect of program design. As we have already remarked, it seems that, in comparison to the novice, the expert is characterized not only by more abstract, hierarchically organized knowledge but also by a broader range of more versatile strategies. Thus experts choose their design strategies on the basis of factors such as the familiarity of the situation, the characteristics of the application task, and the notational features of the language. Novices often experience difficulty not only because of the lack of adequate knowledge but also because of the lack of suitable strategies for responding to a specific situation; thus, in some cases, they may have the necessary knowledge but be incapable of accessing it or using it.

3.3.1 Classification of Strategies

Design strategies can be classified along several axes[5]: top-down vs bottom-up, forward vs backward development, breadth-first vs depth-first, procedural vs declarative.

Top-down vs Bottom-up

A solution may be developed either top-down or bottom-up, that is from the more abstract to the less abstract or vice versa. In the first case the programmer develops the solution at an abstract level and then refines it, progressively adding more and more detail. In the second case, the solution is developed at a very detailed level before its more abstract structure is identified. Top-down development is usually associated with experts while novices typically try to develop bottom-up, by writing directly in the final programming language and then building the abstract structure of the solution. Nevertheless, bottom-up development is used by experts, particularly when libraries of reusable components are available (see Chapter 4) or when a product line is being developed[6].

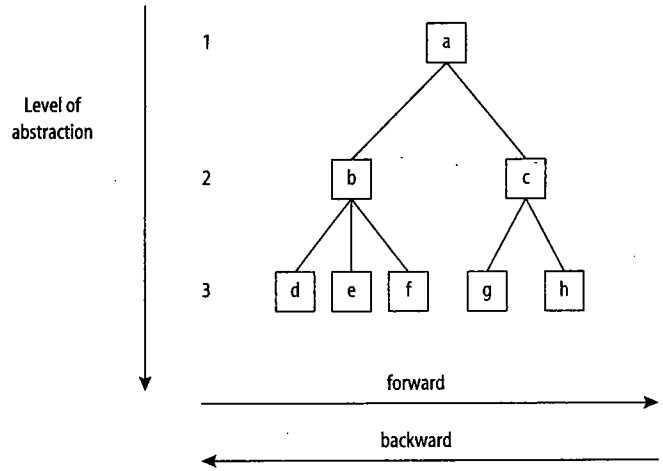

Fig. 3.1 Working out a solution along two axes: abstraction and forward/backward.

Forward vs Backward Development

A design strategy is described as forward development when the solution is developed in direction of execution of the procedure. It is described as backward if it is developed in the direction opposite to that of the execution of the procedure. The use of a forward strategy by beginners often reflects the mental execution of the solution: they rely on this mental execution to work out the solution[7]. In such cases, the development of the solution relies not on computing knowledge but on knowledge of the problem domain: beginners recall known procedures that they develop in a forward sense. Experts also use forward development when they retrieve a known solution schema from memory and implement the elements of the schema in the order in which they will be executed[8]. The direction of development may be backward when no known schema or procedure is available. In Fig. 3.1, a forward strategy would consist in developing level 2 in the order b-c and level 3 in the order d-e-f-g-h.

Breadth-first vs Depth-first

A breadth-first strategy means developing all the elements of the solution at one level of abstraction before proceeding to the next, more detailed, level of abstraction. A depth-first strategy means that one element of the system is developed to all levels of abstraction before any other element is developed. In Fig. 3.1, breadth-first development would mean first developing everything at level 2 (b-c or c-b) and then everything at level 3 (d-e-f-g-h in any order). Depth-first development would mean a-b-d then, e, then f, then a-c-g, then h.

On this axis, it is worth noting that we may distinguish between exploratory activities and development activities. Exploratory activity consists in generating and evaluating alternative solutions in a search space that can also be described at several levels of abstraction. The exploration itself can also be done breadth-first or depth-first.

Wirth's stepwise refinement strategy[9] can be characterized as top-down, breadth-first. The problem is broken down into sub-problems and, at a given level of abstraction, the solution is explored and developed completely[10]. The term 'balanced development' has been used to describe this situation, when the solution is developed completely at level n, then at level $n + 1$, and so on until the least abstract level, that of code, is reached. Experts are observed to use such a strategy to solve problems that are relatively simple and familiar.

Procedural vs Declarative

The development of a solution is said to be procedural when it is the structure of the procedure that controls the solution; the solution is then based on aims or procedures. The development is said to be declarative when static properties, such as objects and roles, control the solution.

This distinction is similar to, but more general than, the distinction between procedure-driven and data-driven software development methods. Methods that concentrate on data analysis and database design would thus be described as declarative, while methods that emphasize functional decomposition would be described as functional.

Mental Simulation

A number of other strategies have been described, notably mental simulation. Simulation can be used to evaluate a solution. In fact, designers often use mental simulation on a partial or complete solution at a higher or lower level of abstraction or on passages of code that they are seeking to understand. Simulation provides a way of verifying that a solution meets the desired objectives and a way of integrating partial solutions by controlling their interactions.

3.3.2 Triggering Conditions

As we have said, it is important to investigate the circumstances under which particular strategies are triggered.

Cognitive Dimensions

Psychologists have studied the notational structure of programming languages and the features of programming environments by identifying different characteristics, called cognitive dimensions[11], that have an effect on the programming activity. Several such dimensions have been identified, among them the following:

hidden dependencies: these arise when a relationship between different items of information is explicit in one direction but not the other. The infamous 'goto' statement is an example of a hidden dependency. It is easily possible to follow this in a forward direction, that is, in the order of execution, but not in the backward direction. This makes some information difficult to find, for example, the conditions under which particular statements are executed;

consistency: this is the property of syntactic and lexical uniformity. For example, the syntax of an integer expression, whether it is a value to be assigned or an array subscript, should always be the same; this was not always the case in early programming languages, such as FORTRAN 2. Consistency makes learning a language or an environment easier;

premature commitment: this occurs when programmers are obliged to take certain decisions before they have all the information necessary. Ordering constraints in some programming environments oblige the programmer to define certain items first, for example, it may be necessary to define a class before using it in the program;

redundancy: this feature may make a program easier to understand but it may make coding longer and more difficult;

resistance to change ('viscosity'): this refers to the amount of effort necessary to make a modification. Thus, a modification at one place may oblige the programmer to make other changes. For example, in Pascal, the need to introduce an extra variable to store a temporary result will lead to the need to add a declaration of the variable in the declaration Section of the code. In general, the more redundant the notation, the more resistant programs are to change;

progressive evaluation: can a partly completed program be tested or evaluated in some other way? Clearly this property helps with reviewing design and code;

'role expressiveness': can the reader easily see what is the relationship between each component of the program and the whole?

visibility: is each part of the program simultaneously visible? Or is it possible at least to display two parts of the program side by side?

The first five of these are primarily programming language issues while the last three are environment issues; however, there is significant interaction between the language and the environment so that, for example, an environment can alleviate the resistance to change evinced by a programming language, by showing the programmer where consequential changes need to be made.

The Notational Structure of Programming Languages

Cognitive dimensions allow us to characterize certain aspects of the notational structure of programming languages. Studies have shown that these dimensions have an effect on the design strategies that the designers follow; there is some evidence to suggest that the effect is greater for designers with an intermediate level of experience than for either experts or beginners[12].

Consider the case of Pascal and Basic. Pascal is much more resistant to change than Basic, because a change in one place is likely to require many other modifications to the code. This is due, *inter alia*, to Pascal's high level of redundancy; every variable must be declared, for example, which is not the case in Basic.

An empirical study[13] has shown that if the notation exhibits strong resistance to change (Pascal in comparison to Basic), the designer tries to minimize the modifications to be made to the code during the design and coding. The Pascal programmer thus uses a top-down strategy based on successive refinement, with frequent taking of notes. The Basic programmer adopts a much more linear and progressive strategy, which leads to frequent backtracking when additions or modifications are to be made.

It may not, however, be the characteristics of the programming language that lead to this difference. Pascal programmers will probably have been taught the language in an academic environment, in which successive refinement is advocated as the 'correct' approach to program design; furthermore most text books on Pascal take this approach. Basic programmers are much more likely to be self taught and thus to adopt a less sophisticated approach; Basic text books are much less likely to advocate successive refinement.

Features of Programming Environments

Cognitive dimensions have also been used to characterize programming environments. Again, studies have shown that the cognitive dimensions of the programming environment affect the strategies adopted by designers. One such study[14] evaluated the object-oriented CO2 language and its environment. This environment has strict ordering constraints, which lead to premature commitment. For example, it is necessary to declare a class before being able to refer to it in a method. This feature of the environment very clearly requires a top-down design strategy since the designers define the classes at an abstract level before developing the code of the methods and using the classes. The study also showed that such a strategy, enforced by the features of the environment, was the cause of errors and that much backtracking was then necessary to modify the erroneous or incomplete class model.

There is a clear conflict here between generally accepted software engineering principles, which emphasize the importance of the top-down approach and declaration before use in producing quality software, and the ease with which small programs can be written and understood.

Problem Type

The type of problem also affects the choice of strategy[15]. A problem can be forward or backward, procedural or declarative, just as a strategy can be. A declarative problem is characterized by a strong data structure (whether of input or output data) data. The program structure is strongly linked to the structure of the data, which provides reliable guidance in the search for a solution. In such a case, the designer will adopt a declarative strategy. If it is the structure of the input data that is strong, the strategy will be declarative and forward. If it is the structure of the results that is strong, the strategy will be declarative and retrospective.

A procedural problem is characterized by a complex procedural structure. The structure of the data gives little help in solving such problems and it is the procedural structure that will guide the design activity. The designer will thus adopt a procedural strategy.

This classification has recently been refined within the framework of object-oriented design[16]. In this paradigm, the declarative/procedural classification does not appear sufficient. A new dimension has been identified: this characterizes not only the structure of the data (or objects in OO) and the structure of the procedure, but also the way in which the two are associated. The strategy adopted by experts is declarative when the problem exhibits a hierarchical solution structure with vertical communication between the objects, while the strategy is procedural when the problem exhibits a flat solution structure with horizontal communications between the objects. In the first case, the procedural structure can be fairly easily deduced from the object structure, which is not true in the second case.

The relationship between the problem and the strategy illuminates, and is illuminated by, a consideration of structured development methods. The use of the structure of the input data to guide the structure of the program is a fundamental idea that goes back to Jackson Structured Programming[17]. The distinction between problem types also explains why the search for a universal development method is doomed to failure.

Internal Factors

Some internal determinants are linked to the computing expertise of the subject and to the programming schemas constructed in his or her memory[18]. Thus experts can follow a top-down, forward strategy for problems that are familiar to them and for which they already have a schema, while the novice will rather follow a bottom-up, retrospective strategy. Empirical studies[19] show that experts use the strategy of successive refinement, i.e. top-down, breadth-first, for familiar problems of reasonable size.

3.4 Organization-centred Approach

For studying the organization of the design activity, two contrasting types of model can be used: the hierarchical model, based on normative models inspired by programming methods, and opportunistic models, based on the results of empirical studies, which emphasize how real activity deviates from a strictly hierarchical model.[b]

3.4.1 Hierarchical Models

The hierarchical model has been strongly influenced by structured programming. According to this model[20], the process of breaking down a problem into a solution is essentially top-down and a breadth-first search for a solution is preferred. All the goals (or functions) of the solution are identified at a certain level of abstraction before being refined successively to more and more detailed levels. This refinement consists either in detailing the abstract elements of the solution or in choosing to deploy more general functions. The prescribed process consists in working at one level of abstraction at a time, preferring to develop the solution breadth-first rather than depth-first.

3.4.2 Opportunistic Models

These models take account of observed deviations from the hierarchical plan and of the conditions that trigger such deviations[21]. While designers seem to accept

[b]Although this contrast between hierarchical design and opportunistic design is challenged by some authors, the importance of opportunistic deviations in professional activity can certainly not be called into question.

the hierarchical model as the paradigm for what they are doing, frequently because this is how they have been taught, they often behave very differently. Real design is organized opportunistically: a bottom-up approach is used as often as a top-down, and depth-first searching may be used in preference to breadth-first, depending on the situation. Thus the hierarchical structure reflects the result of the activity but not the organization of the activity itself.

The studies cited bear witness, on the one hand, to the focusing on different aspects of the solution or on different sub-problems that takes place in the course of the design activity, and, on the other hand, to the possible use of different design strategies to handle different aspects of the process. A solution of hierarchical type is developed by jumping between different levels of abstraction. For example, designers sometimes give priority to refining certain aspects of the solution that they deem critical[22]. This is particularly useful for detecting potential interactions with other parts of the solution.

The adoption of an opportunistic strategy can be triggered by several different factors such as resource limitations or a process of 'meta-cognitive management'.

The plan that guides the design process is hierarchical and opportunistic episodes can be triggered by failures of working memory[23]; information about the hierarchical plan and the way it is being followed is lost when the capacity of the working memory is exceeded. This effect has been demonstrated by asking subjects simultaneously to design and describe orally what they are doing; this double task overloads the working memory, which produces opportunistic deviations in the design process.

Opportunistic deviations are controlled at a meta-cognitive level by criteria that allow the cost of alternative actions to be calculated[24]. One of the reasons that lead the designer to deviate from a hierarchical plan may be the economic utilization of the resources available. If information needed to handle one aspect of the solution is not available, the subject puts it to one side. On the other hand the economic utilization of the information available can lead to certain aspects of the solution at different levels of detail being treated prematurely in comparison with the ideal hierarchical plan of the process. The real process of design will thus be out of phase with the ideal model, either lagging or leading it. Notes made by designers during the process often testify to these phenomena.

3.4.3 The Iterative Nature of Design

Another characteristic of the program design activity is its iterative nature, involving cycles of designing, coding and revising. The cycles are generally accompanied by intensive note-taking.

While the approaches presented above put the emphasis on the parallel with problem solving models, another interesting parallel is with the production of natural language texts. The model of Hayes and Flower[25], one of the most influential cognitive models of text production, identifies three phases in the process of drafting a text:

- planning the structure of the text: the organization of the ideas is a function of domain knowledge and the setting of goals depends on the communication objectives;

- translating the plan of the text into a linguistic representation;
- reviewing the text.

An important feature of this model is that the global process is iterative rather than strictly linear. Further, the ordering of the phases is not strictly predefined[c]. In effect, the organization of the phases depends on the writer's strategy. One effect of the non-linear character is that the writer generates many intermediate results, planning notes, temporary text, additions, etc. The writer also revises versions of the text in such a way as to improve its clarity, its coherence and/or to make the development of the ideas more convincing[26]. Even if the first version of the text is, in general, comprehensible, the revised version accords better with the writer's objectives.

Program design also includes planning, translation and reviewing phases (usually called problem solving or analysis, coding or implementation, and review). Planning demands both the retrieval of knowledge relevant to the solution of the problem and the construction of an abstract solution. The process of translation is equivalent to the implementation of a solution in a specific programming language. Finally, the review process can involve revising the implementation, the abstract solution or the problem representation. It has been shown that cycles including these three phases are produced in the course of software development[27]. The review process leads to changes that may be stylistic, strategic or tactical. Stylistic changes involve changes only to the coding, often to make the program comply with the standards of the organization, that is, its rules of discourse, while strategic and tactical changes involve revision at the level of the abstract solution.

Like writers, program designers make many notes. This is linked to the iterative nature of the process and to the limited capacity of the working memory, whence the need for an external memory. It is also linked to the need to for maintenance programmers to be able to understand why a particular design was adopted.

Software designers produce a large number of notes in natural language while they are designing their programs: design notes, temporary comments or comments that will remain in the final program[28]. Even though the final goal of the design process is to produce a list of instructions coded in a programming language, information expressed in natural language forms an integral part of the design process and of the solution produced.

Various authors[29] have analysed the nature of the natural language information produced during design. Designers use a design notebook to make notes in temporary memory in the course of design. Designers produce notes at different levels of abstraction, from justification of high level design decisions down to implementation details. If they are asked to produce only one type of note on a particular field of the environment, they often deviate from the prescribed category and mix in several types of information.

This behaviour is interpreted as reflecting the iterative and interactive nature of the design process[30]. The different types of information produced are narrowly connected to the designers' representations not on the basis of their syntactic or

[c] In this sense, the model incorporates the opportunistic aspect of design organization.

semantic status, but on the basis of the sub-problem that is being treated at a given moment. Some notes serve simply as external memory and will be deleted subsequently, e.g. the consequence that a design decision has on a sub-problem that will be dealt with later. Other note, e.g. the rationale for certain design decisions, will remain in the program or its documentation as an aid to the human reader.

This intense notetaking activity, which one can describe as the use of external memory, seems to be linked to expertise and has been analysed[31] within the framework of display-based problem solving. It should be noted that, once it is recorded externally, the code itself can be considered as forming part of an external memory[32].

3.5 Modelling the Expert

What distinguishes the expert from the novice? What are the stages on the road to acquiring expertise? Can we distinguish several types of expertise? Research is able to provide at least some answers to questions of this type.

3.5.1 What Distinguishes an Expert from a Novice?

Organization of Knowledge

In the field of computing, as in other fields such as chess or physics, experts form abstract, conceptual representations of problems while novices represent the problems in terms of surface features. Experts possess hierarchically organized knowledge, which gives them a better processing capacity than novices. This is consistent with the knowledge centred approach. Two experimental paradigms have allowed the organization of knowledge by experts and novices in the programming field to be compared: the understanding/recall paradigm and the categorization paradigm.

The understanding/recall paradigm is illustrated by an experiment[33] in which subjects were presented with programs, some of which had the lines of code in the correct order and some of which had the lines in random order. The subjects were then asked to recall the programs. The recall of the experts was better than that of the novices when the lines were in the correct order but not when they were in random order. In the second case, no significant structures could be identified.

The categorization paradigm consists in presenting to programmers a certain number of programs or fragments of programs and asking them to categorize them and explain their categorization criteria. In this way, it can be shown[34] that the categories formed by expert programmers are different from those formed by novices. While the categories formed by the novices depend on the surface features of programs, e.g. the syntactic structure, the categories formed by the experts revealed deeper structures such as functional or procedural similarity.

The multi-dimensional nature of program design expertise needs to be underlined[35]. Knowledge in the following areas is important: general computing knowledge, specialized design schemas, software engineering techniques, knowledge of more technical domains such as knowledge of programming environments, and knowledge of the application or problem domain. On this last point, there has been little work done on expertise in the application domain and its role in programming activities. Some recent studies have shown the importance of such knowledge both in the design and the understanding of programs[36].

Strategies and Use of Knowledge

A complementary approach to the characterization of expertise is to consider whether experts are distinguished not only by possessing more and better organized knowledge than novices but also by their better capacity to use the knowledge[37] they have. This is consistent with the strategy-centred approach and with the organization-centred approach.

Research has shown that, compared with novices,

- experts construct a more complete problem representation before embarking on the process of solving it[38];
- they use more rules of discourse[39]. Recall that such rules govern the instantiation of knowledge schemas during design (see Section 3.2.5);
- they use more meta-cognitive knowledge about programming tasks and about suitable and optimal strategies for completing them[40]. They know a number of possible strategies for completing a task and are able to compare them with a view to selecting the optimal strategy for that task;
- they are capable of generating several alternative solutions before making a choice[41];
- they use more external devices, particularly as external memory[42];
- their design strategy is top-down and forward for familiar and not too complex problems[43], while that novices is bottom-up and backward;
- some aspects of programming tasks are carried out completely automatically[44].

3.5.2 Can Different Levels of Expertise Be Distinguished?

A number of writers[45] have remarked upon the existence of 'super experts' or 'exceptional designers' in software development teams. What distinguishes these super-experts, as they are known by their peers, from other experts? The following characteristics have been identified:

- a broader rather than longer experience: the number of projects in which they have been involved, the number and variety of the programming languages they know;
- technical and computing knowledge;
- a particular ability to combine computing knowledge with knowledge of the application domain;
- social skills such as communication and co-operation.

3.5.3 What Are the Stages in Acquiring Expertise?

Three stages have been recognised in building up knowledge schemas[46]:

- construction of elementary programming schemas;
- construction of complex programming schemas;
- converting the internal structure of schemas into a hierarchy by abstraction from the focal point.

Knowledge of the roles or structural schemas is acquired very early and provides the skeleton on which knowledge schemas are constructed.

Experienced programmers retrieve complex schemas when they have relatively simple problems to program, while novices build these schemas. The retrieval of schemas manifests itself in the fact that the actions of the schema are developed in the canonical order of the schema, which is also the order of execution. In other words, when the programmer has a general solution schema, the program is constructed top-down and in a forward direction.

Internal restructuring of schemas takes place during the acquisition of expertise. From this angle, expertise is built up through the development of hierarchically structured schemas; the focus, that part of the schema that directly implements its goal, becomes hierarchically superior to the other parts of the schema. The focus is then a more prominent part and thus more readily available in memory. In a preliminary scan of memory, depending on the level of expertise, this part is more rapidly recognized than other parts of the schema. In fact, this effect is observed among experts, but not among novice or intermediate programmers.

This work is limited because into fails to take into account the acquisition processes that lead to changes in the organisation of knowledge in the course of acquiring expertise, e.g. the processes of generalization, specialization and proceduralization[47]. Further, it does not take account of the acquisition of knowledge about the conditions under which a schema is applicable. Given a problem, such knowledge allows the designer to select a number of candidate schemas and choose between them. This aspect of acquiring expertise has been studied in tasks such as editing[48] but not in programming.

3.6 Making Tools More Suitable for Programmers

Implementation and Visualization of Schemas

Structural schemas and knowledge schemas can be provided to the designer as aids to program design.

In no case does the notion of a structural schema reduce to the purely syntactic aspects of a language. Present day structural editors are, however, based on the syntax of a language, and often on its control structure. Different levels of nesting in the control structure are distinguished, which, in software engineering, provides a basis for assessing the complexity of a program. From an ergonomic point of view, it would be more useful to provide tools based on the notion of a structural schema. The tools might operate on two levels:

- at the level of a knowledge base. This would provide structural schemas suitable for certain languages or certain programming paradigms. These structural schemas could be provided on demand, to help in structuring programs. For example, to help in designing object-oriented programs, we might provide structural object schemas, associating with a generic object the following generic roles and functions: creation, initialization, read access, write access, inputs, outputs;
- at the level of visualization of these structural schemas in a program being written. This would allow the program to be described at a level relatively independent of the syntax and control structure of the language.

Knowledge schemas could also be made available to designers, to assist them in their search for solutions. This could again be done on two levels:

- the knowledge base. This would provide knowledge schemas suitable for handling certain problems[d]. Taking inspiration from the approach in terms of programming schemas, described in Section 3.2.2, one could formalize both the elementary programming schemas and the complex ones, and characterize the different types of links (composition, specialization, and use) between them. It would be useful to add:
  - information on the use and validity conditions of the programming schemas, e.g. limiting cases;
  - examples of instantiation of the schemas;
  - alternative schemas for the same problem, taking us back to the specialization and use links;
- visualization of programming schemas in a program being written. It is a characteristic of complex programming schemas that they are instantiated in a dispersed manner. In other words, the instructions belonging to a single schema are not contiguous in the program but can belong to different parts. For example, the initialization line and the update line of a counter variable plan are not contiguous in the program code. This feature may explain certain programming errors, especially ones due to forgetfulness. A tool to display instances of schemas[e] would be particularly useful since it would allow semantically connected but non-contiguous elements of the code to be grouped together visually.

More generally, it seems very useful to allow designers to visualize the different types of relation that exist in a program as well as putting them into correspondence one with another. From this point of view, it has been suggested that software development environments should support multiple views of the system, such as control flow, data flow, and functional decomposition. We shall return to this point in Chapter 7, on understanding programs.

---

[d]The work on design patterns (Gamma, Helm, Johnson and Vlissides, 1994) seems a promising way forward here.

[e]The plans representing elements of a single schema could be traced and identified according to Rist (1994), on the basis of data flow analysis. But this means that the program must be complete and not in the course of construction.

Implementation of Design Strategies

We have identified different dimensions that characterize design strategies: bottom-up vs top-down development, forward vs backward, procedural vs declarative. It has been suggested[49] that support might be provided for strategies varying according to the first two of these dimensions. While top-down strategies are often supported – or indeed, enforced – by programming environments, bottom-up strategies and their linking with top-down strategies are rarely, if ever, supported. As far as the second dimension is concerned, support for backward development is provided by certain environments such as MAIDAY[50] but support for a forward strategy seems much more difficult to implement, because the goals of the designer are more difficult to infer than the prerequisites of explicitly stated goals. On this last point, it seems to us possible to envisage some support for a forward strategy, provided that it is not based solely on the code but uses also a more abstract definition of the solution, produced by the designer. Further, some help with simulation should be provided because this activity, particularly useful for debugging (see Chapter 7), is often a source of errors.

Opportunistic Design

The emphasis on the opportunistic nature of design has made the designers of programming environments conscious of the serious constraints that they place on programmers by imposing a top-down approach. Recent articles[51] bear witness to the desire to allow for design organized in a more opportunistic fashion. This means, on the one hand, not restricting the development to being completely top-down and breadth-first, and, on the other hand, supporting opportunistic activity through tools that allow the solution plan to be managed so as to warn of errors and omissions caused by opportunistic deviations. We can illustrate these two aspects by considering two tools, HoodNICE[f] and ReuseNICE[52], intended to support design and reuse. They are based on the HOOD (Hierarchical Object Oriented Design) method, which is purely top-down, but allow the following opportunistic deviations:

- the breadth-first decomposition strategy defined by the HOOD method can be interrupted to allow tactically better choices to be followed;
- the HoodNICE editor was modified to allow bottom-up development as well as top-down. In particular, a child object can be created before its parent;
- several levels of the design tree can be displayed and edited at the same time;
- the solution can be temporarily inconsistent with the rules defined by the method (e.g. naming rules, order of definition of code entities). An off-line checker is available when required, to check for inconsistencies;
- notes concerning, for example, consequences of decisions taken and how they affect later tasks can be made and stored in a work book.

[f]HoodNICE was developed by Intecs Sistemi and its ergonomic evaluation was carried out as part of the European SCALE (System Composition and Large Grain Component Reuse Support) project. See Oquendo, Détienne, Gallo, Kastner and Martelli, 1993.

Finally, it has been suggested that, in order to support opportunistic activity, the designer might be provided with a representation of the solution plan, which would help to identify incomplete tasks or tasks put on one side as a result of side tracking. Such support can be envisaged at the level of the process model, which would make it relatively independent of the programming environment.

Teaching Tools

A learning model that takes account of the process of acquiring knowledge schemas, as in Soloway's approach, has given rise to teaching tools based on the idea of programming schemas. The underlying hypothesis is that presenting and teaching the schemas to beginners will help them to develop expertise[53].

As a complement to this, according to an acquisition model in terms of internal restructuring of schemas, one might think that practice with structured programming and, more generally, training in design would facilitate the development of expertise. This would focus beginners' attention on the important parts of the schema and thus encourage them to restructure the schema hierarchically.

While the validity of the first approach has still to be conclusively proved, the second has hardly been tested experimentally[g]; the one study of which we are aware[54] shows that, for a given amount of practical experience, training in structured programming leads to better performance than a more classical training. One limit of these approaches is that they do not help to build strategic knowledge, which is just as important as generic knowledge of solutions.

3.7 Future Research

One direction for future research is the analysis of the subjects' understanding of the problem to be solved. In every study of program design, the same implicit assumption is made: the subjects will all have constructed the same representation of the problem. This is manifestly false when the problem to be solved is non-trivial. Much of the variability between individuals observed in studies of the design process might be explained by this.

Another direction is to analyse further the design strategies and the organization of the design activity, and to take more detailed account of the conditions under which they are adopted. From a theoretical point of view, the objective would be to build a model that integrates the different design approaches. We have already seen that they are, in practice, complementary and this fact deserves to be more widely recognized and better articulated.

Finally, so far as the acquisition of expertise is concerned, the mechanism for constructing knowledge at present envisaged is extremely simplified since it is limited to a mechanism for acquiring knowledge schemas. It would be useful to go further in the analysis of these acquisition mechanisms. The analysis of the

[g]Both approaches are, of course, widely used in teaching programming and software design and widely believed to be superior to other approaches. All that is being asserted here is that there is little scientific evidence, based on properly designed experiments, to demonstrate this superiority.

mechanisms for reorganizing this knowledge as well as of the mechanisms for constructing knowledge about schemas are the path along which research might proceed.

References

1. Hoc, 1984a; 1987b.
2. Adelson, 1981, 1984; Black, Kay and Soloway, 1986; Détienne, 1990b, 1990c. Détienne and Soloway, 1990; Rist, 1986, 1989, 1991; Robertson, 1990; Soloway, Erhlich and Bonar, 1982a; Soloway, Erhlich, Bonar and Greenspan, 1982b; Soloway and Ehrlich, 1984.
3. Soloway, Erhlich and Bonar, 1982a.
4. Soloway and Ehrlich, 1984.
5. Visser and Hoc, 1990.
6. See, for example, the papers from the two product line sessions in ICSE (2000).
7. Hoc, 1987b.
8. Rist, 1991.
9. Wirth, 1974.
10. Adelson and Soloway, 1985.
11. Green, 1989, 1990; Green and Petre, 1996.
12. Davies, 1991.
13. Green, Bellamy and Parker, 1987.
14. Détienne, 1990a; 1990d.
15. Hoc, 1981.
16. Détienne, 1995; Chatel and Détienne, 1996.
17. Reference required.
18. Rist, 1991.
19. Adelson and Soloway, 1985, 1988; Carroll, Thomas and Malhotra, 1979; Guindon and Curtis, 1988; Jeffries, Turner, Polson and Atwood, 1981.
20. Adelson and Soloway, 1985.
21. Guindon, Krasner and Curtis, 1987; Guindon, 1990a; Visser, 1987.
22. Jeffries, Turner, Polson and Atwood, 1981; Visser, 1987.
23. Davies and Castell, 1994.
24. Visser, 1994a, 1994b.
25. Hayes and Flower, 1980.
26. Bereiter, Burtis and Scardamalia, 1988.
27. Gray and Anderson, 1987.
28. Bellamy, 1994a; Davies, 1996; Henry, Green, Gilmore and Davies, 1992.
29. Détienne, Rouet, Burkhardt and Deleuze-Dordron, 1996; Rouet, Deleuze-Dordron & Bisseret, 1995a, 1995b.
30. Détienne, Rouet, Burkhardt and Deleuze-Dordron, 1996.
31. Davies, 1996.
32. Green, Bellamy and Parker, 1987.
33. Shneiderman, 1976.
34. Adelson, 1981, 1985.
35. Guindon and Curtis, 1988.
36. Shaft and Vessey, 1995; Sharp 1991.
37. Davies, 1993a; Gilmore, 1990.
38. Batra and Davies, 1992.
39. Davies, 1990a.
40. Eteläpelto, 1993.
41. Guindon, Krasner and Curtis, 1987; Jeffries, Turner, Polson and Atwood, 1981; Visser, 1994a.
42. Davies, 1996.
43. Adelson and Soloway, 1988; Rist, 1991.
44. Wiedenbeck, 1985.
45. Curtis, Krasner and Iscoe, 1988; Sheppard, Curtis, Milliman and Love, 1979; Sonnentag, 1995, 1996.
46. Davies, 1994; Rist, 1991.
47. Rumelhart and Norman, 1978.
48. Black, Kay and Soloway, 1986.
49. Visser and Hoc, 1990.

50. Guyard and Jacquot, 1984; Hoc, 1988.
51. Bisseret, Deleuze-Dordron, Détienne and Rouet, 1995; Guindon and Curtis, 1988; Guindon, 1992; Visser and Hoc, 1990.
52. D'Alessandro and Martelli, 1994.
53. Anderson, Boyle, Farrel and Reisner, 1987; Bonar and Cunningham, 1988.
54. Davies, 1990b, 1993b.