8 The Processes Involved in Designing Software

Robin Jeffries, Althea A. Turner, Peter G. Polson
University of Colorado

Michael E. Atwood
Science Applications, Inc., Denver

INTRODUCTION

The task of design involves a complex set of processes. Starting from a global statement of a problem, a designer must develop a precise plan for a solution that will be realized in some concrete way (e.g., as a building or as a computer program). Potential solutions are constrained by the need to eventually map this plan into a real-world instantiation. For anything more than the most artificial examples, design tasks are too complex to be solved directly. Thus, an important facet of designing is decomposing a problem into more manageable subunits. Design of computer systems, software design, is the particular design task to be focused on in this chapter.

Software design is the process of translating a set of task requirements (functional specifications) into a structured description of a computer program that will perform the task. There are three major elements of this description. First, the specifications are decomposed into a collection of modules, each of which satisfies part of the problem requirements. This is often referred to as a modular decomposition. Second, the designer must specify the relationships and interactions among the modules. This includes the control structures, which indicate the order in which modules are activated and the conditions under which they are used. Finally, a design includes the data structures involved in the solution. One can think of the original goal-oriented specifications as defining the properties that the solution must have. The design identifies the modules that can satisfy these properties. How these modules are to be implemented is a programming task, which follows the design task.

This chapter presents a theory of the global processes that experts use to control the development of a software design. After a review of some relevant

literature, the theory is described in detail. Thinking aloud protocols collected from both expert and novice designers on a moderately complex problem provide evidence for these theoretical ideas. Finally, we speculate on how such processes might be learned.

RESEARCH ON DESIGN AND PLANNING

Although there has been little research that focuses directly on problem-solving processes in software design, there are a number of research areas that are peripherally related. The first of these, formal software design methodologies, is indicative of the guidelines that experts in the field propose to structure the task of designing. The second area, automatic programming, provides a detailed analysis of the task from an artificial intelligence viewpoint. Finally, research on planning and design gives insight into planning processes that may be general across domains.

Software Design Methodologies

There are two reasons for considering the professional literature in this field. A reasonable model of performance in any domain ought to relate to accepted standards of good practice in that domain. These formalized methods were written by experts in the area trying to convey to others the procedures they use to perform the task. In addition, most expert designers are familiar with this literature and may incorporate facets of these methodologies into their designs.

Software design involves generating a modular decomposition of a problem that satisfies the requirements described in its specifications. Design methods provide different bases for performing modular decompositions. There are two prevailing views in the literature as to what this basis should be. Both positions prescribe problem reduction approaches to the design process. One focuses on data structures and the other on data flow. The various methodologies differ in the nature and specificity of the problem reduction or decomposition operators and of the evaluation functions for determining the adequacy of alternative decompositions.

With the data-structure-oriented approaches (e.g., Jackson, 1975; Warnier, 1974), a designer begins by specifying the input and output data structures according to certain guidelines. A modular decomposition of a problem is identified by deriving the mapping between the input and output data structures. Because such methods involve the derivation of a single "correct" decomposition, there is no need for evaluation criteria or the comparison of alternative decompositions.

Data-flow oriented approaches (Myers, 1975; Yourdon & Constantine, 1975) are a collection of guidelines for identifying trial decompositions of a problem.

Thus, these methods are more subjective, allowing a designer to exercise more judgment. As a result, numerous heuristics for evaluating potential decompositions are used with these methods. Examples of such evaluation guidelines include: maximizing the independence and cohesion of individual modules, providing a simple (as opposed to general) solution to the current subproblem, etc. These guidelines control the evaluation of possible solutions to a design problem and the generation of new alternative designs.

Most formal software design methodologies require that the design proceed through several iterations. Each iteration is a representation of the problem at a more detailed level. Thus, the initial decomposition is a schematic description of the solution. This becomes more detailed in the subsequent iterations. In general, this mode of decomposing the problem leads to a top-down, breadth-first expansion of a design.

There are competing views that prescribe different modes of expansion. Some of these are characterized by such terms as *bottom-up*, *middle-out*, or *inside-out* (Boehm, 1975). Such positions have been developed in response to what some individuals feel are unsatisfactory properties of a top-down expansion. There are problems in which it is necessary to understand certain crucial lower-level functions in order to identify high-level constraints on the design. These alternative modes of expansion may be used by a designer in problems for which an initial decomposition is difficult to derive. There are undoubtedly problems for which each of these methodologies is particularly suited. However, the formal literature on software design lacks a mapping between types of problems and the appropriate design methodology.

Automatic Programming Systems

Another source of information about the task of software design comes from automatic programming systems. The term *automatic programming* has been used to refer to activities ranging from the design and development of algebraic compilers to systems that can write a program from information given in the form of goal-oriented specifications (Biermann, 1976; Heidorn, 1976). The latter represent attempts to specify the procedures of software design in a mechanizable form.

Simon's (1963, 1972) Heuristic Compiler was one of the earliest proposals for a programming system that generated code from abstract specifications. This program's task was to generate IPL-V code for subroutines that were components of some larger program. It was implicitly assumed that the original specifications had been decomposed into detailed functional descriptions for a collection of routines that would make up the complete program.

The definitions of routines to be generated by the Heuristic Compiler could take one of two forms, with each form being handled by a separate special compiler. The first form involved a before and after description of the states of certain cells in the IPL system. The specification described the inputs and outputs

of a routine. The state description compiler's task was to derive the sequence of IPL instructions that brought about that transformation. The other form of definitions was in terms of imperative statements describing the function to be performed by a given subroutine, which was handled by the function compiler. Both specialized compilers used suitably generalized forms of means-ends analysis to generate sequences of IPL instructions that would meet the input specifications.

One branch of current research on automatic programming can be viewed as attempts to generalize the ideas that were originally contained in the state description compiler. Biermann (1976) describes several automatic programming systems that derive programs from examples of input-output behavior for a routine or from formal descriptions of inputs and outputs. Note that the data-structure-oriented software design methodologies discussed earlier resemble these systems in their focus on deriving detailed actions from inputs and outputs.

Other automatic programming systems have been developed that generate routines from information supplied through a natural language dialogue with the user (Heidorn, 1976). These efforts can be viewed as generalizations of the function compiler. Such systems consist of four components (Balzer, 1973; Green, 1977; Heidorn, 1976). First, the system acquires a description of the problem to be solved, frequently via interactions with a relatively naive user. Second, this information is synthesized into a coherent description of the program to be written (Green, 1977). This description is then verified, and additional information, if necessary, is acquired through further interactions with the user (Balzer, Goldman, & Wile, 1977). Finally, the refined description is used as input to a subsystem that synthesizes the program in the high-level language, making decisions about data structures, algorithms, and control structures. Much of the current work in automatic programming focuses on the last of these components.

Balzer and his colleagues have considered the task of transforming an informal natural language specification of a program into a formal description of a design. This design would then be input into a code generation subsystem. There are two aspects of Balzer's work that are relevant here. First, he attempts to develop techniques that enable one to carry out the initial phases of the design effort. Incomplete goal-oriented specifications are first translated into abstract, incomplete functional specifications and then refined into a complete set of formal specifications for the program. Second, the knowledge used by Balzer's system is domain independent. This system can be contrasted with the programs of Long (1977) and Mark (1976) that are strongly domain dependent, and where design problems are proposed in a single microworld.

A system that is designed to deal with the problems of detailed design and code generation is a program called PECOS (Barstow, 1977, 1979). PECOS generates LISP code from a high-level description of input and output data structures and the algorithms to be used to solve the problem. A distinguishing

feature of PECOS is that the program uses a collection of rules. It encodes both general knowledge and specific information about LISP to guide its problem-solving efforts, rather than using a uniform strategy like means-ends analysis.

PECOS' knowledge base is in the form of a large set of rules. General rules deal with representation techniques for collections, enumeration techniques for collections, and representation techniques for mappings. Each of these subsets of rules can be organized into a hierarchical structure with a number of intermediate levels between the most abstract concepts (e.g., collection) and information about specific procedures or data structures (e.g., linked free cells).

PECOS employs problem-solving mechanisms that iteratively refine each component of the specifications. A partially refined subproblem is selected, and then a rule is applied to it. Each rule application can produce one of three outcomes. First, the subproblem can be refined to the next lower level of detail. Second, crucial properties of some component of the subproblem can be identified and included in the description. Third, additional information about the subproblem can be gathered.

This review of automatic programming demonstrates that there are two components to the task of software design. The first is the translation of the initial goal-oriented specifications into a high-level functional decomposition of the original problem. This incomplete, abstract description of the problem must then be refined into a set of formal specifications that precisely define data structures, control structures, and the functions performed by various modules in the program. The second stage of the design process involves a collection of implementation decisions. These decisions specify data structures and algorithms that satisfy the functional descriptions and efficiency criteria. The first phase requires powerful problem-solving strategies that can factor the original problem into a collection of subproblems. It also requires the generation of successive refinements of each subproblem, incorporating more and more detail about the developing solution.

Models of Planning and Design

There exist two problem-solving systems (Hayes-Roth & Hayes-Roth, 1979; Sacerdoti, 1975) that contain mechanisms that seem adequate to carry out the processes required in the initial phase of the design process. Both of these systems generate a plan of action.

Sacerdoti's (1975) NOAH solves robot planning problems by a process of successive refinement. Sacerdoti assumes that the knowledge necessary to generate a plan is organized in a collection of knowledge structures, each of which contains the specification of some subgoal and the actions necessary to accomplish that subgoal. Each unit of knowledge has the information necessary to take one element of a developing plan and produce its next more detailed refinement. Sacerdoti assumes that the complete plan is generated iteratively. At any stage of the planning process, each segment of the plan is expanded to its next level of

refinement. Then generalized problem-solving processes, called *critics*, are used to reorganize this more detailed plan into an internally consistent and efficient sequence of actions. The process repeats itself at the next level, terminating with a plan whose individual steps can be executed to solve the initial problem.

Hayes-Roth & Hayes-Roth (1979) describe a HEARSAY-like system that plans routes for performing a collection of everyday errands. Knowledge about the planning of errands is organized into a collection of pattern-directed modules, called *specialists*, that communicate through a global data structure called the *blackboard*. The behavior of this system is opportunistic in the sense that data currently on the blackboard can trigger a specialist that makes a decision at some arbitrary level of abstraction in the developing plan.

Hayes-Roth and Hayes-Roth point out that a system like NOAH is quite rigid, in that it is restricted to a purely top-down, breadth-first expansion of a solution. Their system, in contrast, is capable of making a best or most useful decision at any level of abstraction; is capable of incremental or partial planning; and can adopt different planning methods depending on the specifics of a given problem.

Many of Hayes-Roth and Hayes-Roth's criticisms concerning the rigidity of a program like NOAH are well taken. On the other hand, many of the phenomena that they have observed in their protocols may be due to the task and the level of expertise of their subjects. None of their subjects had extensive experience with errand-planning tasks. It may be the case that one would observe quite different behavior in an environment that required the solution of a large number of subproblems and the integration of these solutions. One might also expect more orderly kinds of behavior in situations where successful performance required the integration and utilization of a large, well-organized body of relevant knowledge.

There has been a limited amount of research on the process of design or on problems that are difficult enough to require the construction of an elaborate plan. Much of the work on expert problem solving in thermodynamics (Bhaskar & Simon, 1977), physics (Larkin, 1977), and other semantically rich domains is not directly relevant to processes involved in solving design problems, because these studies all use problems that can be solved by a single, well-understood problem method, or schema. An expert in these domains first has to identify the relevant schema and then apply the schema to the problem. In contrast, the major task in design is the reduction of the original problem into a collection of subproblems.

Levin (1976) has attempted to develop a theory of software design processes that is consistent with current thinking on the structure of the human information-processing system and known problem-solving methods. Levin (1976) postulates that design can be viewed as involving three fundamental processes: "selecting problems to work on, gathering information needed for the solution, and generating solutions [p. 2]." Levin argues that the problem selection process is controlled by a set of global strategies and local information about

constraints that are directly relevant to the current subproblem. He developed a simulation model that takes as input the protocol of an expert designer working on a fairly difficult problem and produces a list of subgoals generated by that designer during the process of solving the problem.

Simon (1973) sketches out a theory of psychological processes involved a design task in the context of discussing the distinction between well-structured and ill-structured problems.

> The whole [architectural] design then, begins to acquire structure by being decomposed into various problems of component design, and by evoking, as the design progresses, all kinds of requirements to be applied in testing the design of its components. During any given short period of time, the architect will find himself working on a problem which, perhaps being in an ill structured state, soon converts itself through evocation from memory into a well structured problem [p. 190].

Simon's view of the design process is that the original design problem is decomposed into a collection of well-structured subproblems under the control of some type of executive process that carries out the necessary coordination functions. Also note that information retrieved from long-term memory is incorporated into the developing solution; it is this additional information that converts the original ill-structured problem into a collection of well-structured problems.

Much of the work discussed previously focuses on the decomposition of complex tasks into more manageable subtasks. Our interpretation of the literature on software design is that this decompositional process is central to the task. Moreover, we believe that the mastery of decomposition should be what differentiates experts from novices. The theory to be presented next is built on the process of decomposition and its associated control strategies.

A THEORY OF PROBLEM SOLVING IN SOFTWARE DESIGN

The following is an outline of a theory of processes involved in solving a software design problem. The successful performance of this task involves the coordination of a complex set of processes. Some apply abstract knowledge about the task. Others retrieve computer science knowledge or information about the design problem or are involved in the storage of relevant information for later use in solving problems. The focus of this discussion is on the global structure of the design task, particularly its guiding control processes, and on the manipulation of knowledge within the problem-solving effort.

Experts have knowledge concerning the overall structure of a good design and of the process of generating one. Using this knowledge, they direct their actions to insure that their designs will satisfy these structural constraints. This implies that skilled designers have knowledge describing the structure of a design inde-

pendent of its content. This abstract knowledge about design and design processes, along with the set of procedures that implement these processes, will be referred to as the *design schema*. This schema, which develops through experience with software design, enables efficient management of a designer's resources in doing this particular specialized and complex task. We propose that the generation of a design is controlled by the interaction between the design schema and the more specific knowledge that describes how to accomplish particular goals.

A schema is a higher-order knowledge structure that governs behavior in a particular domain or activity, providing a broad abstract structure onto which an exemplar is to be mapped. These knowledge structures specify principal elements of a given domain and include mechanisms that drive the generation process and that lead to outcomes that are structured according to conventions shared by expert members in a discipline. A schema can be used to organize complex material into constituents and may be applied recursively to break some of these constituents down further. These same structures also guide the comprehension process by arranging incoming information so that it is structured according to the underlying abstract schema. Absence of an appropriate schema can interfere with both the initial comprehension and subsequent recall of a text.

The design schema is used in both the generation and comprehension of software designs. The design schema is not tied to any specific problem domain but consists instead of abstract knowledge about the structure of a completed design and the processes involved in the generation of that design. It accounts for the overall structure of expert design behavior and the similarities among experts. Of course, the design schema will differ from expert to expert, because their experiences with software design will not be identical. However, the overall nature of these schemata will be similar for most people. Therefore, we choose to simplify this discussion by referring to a single, modal design schema.

The design schema develops as a result of experience with software design. Originally, a designer's approach to this task is assumed to involve general problem-solving strategies, such as "divide and conquer." As an individual has more and more experience with this activity, these general strategies are transformed into a specialized schema. The schema is developed through the addition of domain-specific concepts, tactics, and evaluative criteria. Whenever a designer's specialized schema is inadequate to solve a problem, more general strategies take over.

The design schema is assumed to include: (1) a collection of components that partition the given problem into a set of meaningful tasks; (2) components that add elements to tasks in order to assure that they will function properly (e.g., initialization of data structures or of loops); (3) a set of processes that control the generation and/or comprehension of designs; and (4) evaluation and generation procedures that ensure effective utilization of knowledge. Each component of the design schema is composed of both declarative and procedural knowledge about

the abstract nature of the design process. The schema can be applied recursively, which leads to a modular decomposition of the problem into more and more detailed modules.

The schema can be viewed as driving the generation of a software design by breaking up the initial task into a set of subproblems. Knowledge of the particular subproblems that are identified during this decomposition interacts heavily with the schema. However, the design schema itself does not contain knowledge about any particular class of problems. The schema can be applied to the original problem or to any subproblem at a lower level. The recursive application of the design schema to subproblems enables decomposition of each problem into a manageable set of tasks.

How the decomposition proceeds depends on the designer, the designer's experience, and the problem at hand. There are several decomposition strategies that a designer can use to guide the process. One strategy is to break the problem into input, process, and output elements. Whereas there are other strategies that could be used to decompose some problems, the input-process-output strategy is preeminently used. In order to keep this discussion more concrete, we describe decomposition in terms of this prevailing strategy.

The initial pass at decomposition results in a representation of the problem that is a simplified "solution model" of the system; that is, a model is devised specifying a set of tasks that will solve the problem and a control structure for these tasks. It is then expanded into a set of well-defined subproblems. The solutions to these subproblems represent a solution to the original design problem. This process of decomposition is now applied to each subproblem in turn, resulting in more and more detailed plans of what should be done to accomplish the task. Once an individual selects a given element to refine further, the schema is assumed to execute to completion, developing a solution model for that element and refining it into a more detailed plan. If any of the elements resulting from this process are complex (i.e., accomplish multiple functions that are not recognized as having known solutions), the schema is called recursively to reduce them to the next level of detail.

The application of the schema to an element of a design causes a set of high-level goals and procedures for accomplishing those goals to be activated. Thus, the schema includes procedures that examine information relevant to the expansion of a given element, critique potential solutions, generate alternative solutions for a subproblem, etc. The input component, for example, finds information that must be passed to a process component before the actual processing can be initiated. If the chosen input data structure is complex, that is, requires some degree of processing itself to generate the appropriate data structure, then a new subproblem is generated as a descendant of the original one.

The design schema represents the global organization of a designer's professional knowledge. As such, it will impact almost every facet of the designer's behavior in the domain. Nevertheless, the design schema does not encompass a

person's knowledge of specific facts in computer science or understanding of how things function in the real world. There are undoubtedly other aspects of this domain that should not be subsumed under the schema, but our theory is not sufficiently developed to isolate them at this point.

The decomposition process uses two additional problem-solving strategies. The first can be described as problem solving by analogy, or, to use Sussman's (1977) term, "the debugging of almost right plans." When the solution model generated for a given subproblem, or some part of it, is recognized as being analogous to an already understood algorithm, that algorithm is evaluated for applicability in the current context. If it is found to be reasonably applicable, it is debugged and incorporated into the developing solution. This attempt to retrieve previous solutions is invoked once a solution model has been derived, but before any further refinement takes place.

The second method can be characterized as problem solving by understanding. This is prominent in cases where an element identified by application of the design schema is not understood in enough detail for the design schema to be applied to it. The designer's knowledge of the problem area in question, as well as of computer science, is then used to refine the understanding of this element. This method may be employed at any point in the solution. It is most frequently applied when developing a solution model but can also be applied during refinement of a subproblem.

In addition to controlling the overall problem-solving process, the design schema has some coordination and storage functions. Successful solution of a design problem requires that information generated during each problem-solving episode be stored in long-term memory. This information must be interconnected with the expert's knowledge about computer science as well as with the developing solution. Much of what goes on can be described as the development of an understanding of the problem. The information generated during these understanding phases must be stored such that it can be retrieved later for the solution of other subproblems. The design schema ensures that successive episodes are organized so information generated can be stored in a coherent representation of the developing solution.

The utilization of memory is influenced by its organization and by the effectiveness of the abstract cues provided by the schema. Experience enables concepts to be linked on the basis of the utility of considering the concepts together. This usefulness can be defined in terms of concepts that frequently occur in the same context (e.g., linked lists and efficient insertion and deletion of items at random places within the list) or that are alternative solution techniques to similar problems (e.g., a symbol table may be represented as a hash table or as a static tree table).

When a computer science concept is learned, that concept is associated with the context in which it is learned. For example, one might first learn about a particular data structure in the context of a certain problem. Later, in another

problem that would be appropriate for this type of data structure, one might fail to apply this new concept, because the current context might not encourage its retrieval. Eventually, through experience with the concept in many other contexts, it becomes linked to more abstract conditions for its use. Further, as a person's design schema develops such that it can manage the complexity of alternate solutions, this concept would become connected to the concepts of other data structures that would be considered in similar contexts. Thus, memory organization is altered, reflecting the designer's developing schema and previous experiences.

The major control processes of the design schema are summarized as a set of very abstract production rules in Fig. 8.1. Each rule encapsulates a complex subprocess that an expert may use while generating a software design. The rules are an attempt to capture the global control processes only; many aspects of the design schema are not addressed at all. In particular, no reference is made to the processes that generate alternative solutions or critique designs, or to the memory coordination functions that the schema performs. Moreover, the rules only refer to the generation of a design; they do not encompass its comprehension.

The goal of software design is to break down a problem into a set of subprocesses that accomplish the task. After the initial decomposition, there may be multiple subproblems to be solved. The designer must have a way of selecting a problem to work on from the currently unsolved subproblems. The selection rule (Rule 1) provides a coherent way of determining what problem to tackle next. The rules assume that the list of unsolved subproblems are stored on an agenda. The selection rule results in one of them being marked as a current subproblem. The other rules are applied to this problem.

The usual order in which a designer attempts subproblem solution is top-down, breadth-first. The design schema causes each element of the current iteration to be expanded to the next level of detail. This expansion continues until a new representation of the complete solution is developed at the next level of detail. Solving the problem top-down, breadth-first ensures that all the information about the current state of the design at one level of abstraction will be available to the next iteration.

One reason for this strategy is that the elements of a developing design can interact with each other. Although one of the heuristics that guides the decomposition process is the attempt to define subproblems that do not interact or interact only weakly, this is not always possible. Further refinement of one element may require knowledge of decisions that will be made in developing a not-yet-considered element.

A designer may choose to deviate from this order. These deviations are dictated by individual differences in design style, in the amount of knowledge that the designer may have concerning the problem, or in differences in the solution model. The solution model with its various constituents may enable a designer to recognize that a solution relevant to the current problem is known.

```
DESIGN SCHEMA RULES:  SELECTION RULE
  DESIGN SCHEMA RULE 1:
    IF      (no current subproblem exists)
              AND (any unsolved subproblems on agenda)
    THEN    (select highest priority subproblem or, if multiple subproblems at
              highest priority, select next subproblem in breadth-first order
              at highest priority and make it new current subproblem)

DESIGN SCHEMA RULES:  SOLUTION MODEL DERIVATION PROCESS
  DESIGN SCHEMA RULE 2:
    IF      (p is current subproblem)
              AND (solution model for p does not exist)
    THEN    (set goal to create solution model for p)
  DESIGN SCHEMA RULE 3:
    IF      (goal to create solution model for p)
              AND (p is not well understood)
    THEN    (retrieve information relevant to p and refine understanding of p)
              AND (add new subproblem p' to agenda)
              AND (make p' current subproblem)
  DESIGN SCHEMA RULE 4:
    IF      (goal to create solution model for p)
              AND (p is understood as "trivial")
    THEN    (assert that p is solved)
              AND (delete p as current subproblem)
  DESIGN SCHEMA RULE 5:
    IF      (goal to create solution model for p)
              AND (p is understood as "complex")
    THEN    (define solution model for p)

DESIGN SCHEMA RULES:  SOLUTION RETRIEVAL PROCESS
  DESIGN SCHEMA RULE 6:
    IF      (solution model for p exists)
    THEN    (search memory for potential solutions which match critical fea-
              tures of solution model for p)
  DESIGN SCHEMA RULE 7:
    IF      (potential solution s to problem p is found)
    THEN    (evaluate applicability of s)
  DESIGN SCHEMA RULE 8:
    IF      (potential solution s to problem p is highly applicable)
    THEN    (assert that p is solved)
              AND (delete p as current subproblem)
  DESIGN SCHEMA RULE 9:
    IF      (potential solution s to problem p is moderately applicable)
    THEN    (add to agenda new subproblem p' created from solution model
              for p augmented by s)
              AND (make p' current subproblem)
```

FIG. 8.1. A production system representation of the design schema control processes.

see (A) previous page.


```
  DESIGN SCHEMA RULE 10:
    IF      (potential solution s is weakly applicable)
    THEN    (reject potential solution s)

DESIGN SCHEMA RULES:  REFINE SOLUTION MODEL DECOMPOSITION
  DESIGN SCHEMA RULE 11:
    IF      (no potential solution to problem p is found)
    THEN    (expand solution model for p into well-defined subproblems using
              understanding and evaluation processes as needed)
              AND (add each new subproblem generated to agenda)
```

FIG. 8.1. (*Continued*)

This solution then can be adapted to the current situation. Also, the representation of each element of the solution model may enable a designer to estimate their relative difficulties or to identify potential interactions that impact further development of the design. The realization that one or more constituents have known solutions, are critical for success, present special difficulties, etc. can cause the designer to deviate from a top-down, breadth-first expansion of the overall design by assigning a higher priority to a particular constituent.

Once a subproblem has been selected, the designer attempts to derive a solution model for it (Rule 2). Recall that the solution model is an abstract simplified description of elements of the subproblem's solution. This solution model is the basis for all succeeding work on this problem. Rules 2 through 5 describe the processes that may result in the generation of the solution model for the current subproblem. If the current subproblem is perceived to be complex, the designer must first undertake to reformulate it before a solution model can be generated. Rule 3 represents the process by which information relevant to the subproblem is considered, and a new more understandable problem is produced. Once it is precisely formulated, a solution model is generated if the problem requires further decomposition (Rule 5). If the problem, once understood, is sufficiently simple, it is marked as solved and is not further considered (Rule 4).

The next set of rules (Rules 6 through 10) encompass the processes by which a designer attempts to retrieve from memory a previously constructed solution to all or part of the current subproblem. First, the solution model for this problem is used as a retrieval cue to access potential solutions in memory (Rule 6). These solutions are then evaluated for their usefulness in the current context (Rule 7). The rules give a simplified characterization of the results of this evaluation process. The solution is either accepted as is (Rule 8), modified to fit the current situation (Rule 9), or rejected (Rule 10).

If no usable solution to the current subproblem is found, the solution model is refined into a collection of well-defined subproblems (Rule 11). This refinement process takes into account data flow, functional analysis, aesthetic, practical, and other criteria, and implementation considerations. Each new subproblem thus

generated is added to the agenda. The set of rules is applied to the subproblems on the agenda until all problems are considered to be solved.

The theory just presented describes a mechanism by which experts are able to integrate and structure their high-level knowledge of software design. Although experts in the field should manifest mature design schemata, we would not expect beginning designers to show evidence in their behavior of this complex organization. Therefore, many differences we might observe between experts and novices can be attributed to differences in the state of development of their design schemata.

A COMPARISON OF EXPERT AND NOVICE DESIGN PROCESSES

The processes involved in designing software are learned through experience. To examine their development, we collected thinking-aloud protocols from people at various skill levels. This set of protocols forms a rich data base of evidence about the problem-solving processes used in software design. There are, of course, many similarities in the way experts and novices approach this process; subjects at different levels used many of the same global processes. Differences as a function of expertise fall into two major categories: the processes used to decompose the problem and solve individual subproblems, and the representation and utilization of relevant knowledge. In this section, the similarities and differences among subjects are discussed and related to the theoretical ideas proposed earlier.

Subjects and Materials

Four of the subjects were experienced designers. They include a professor of electrical engineering (S35), two graduate students in computer science (S2 and S5), both of whom had worked as programmers and designers for several years, and a professional systems analyst with over 10 years experience (S3).

The five novices were undergraduate students recruited from an assembly language programming class. They had all taken from four to eight computer science courses; most had had part-time programming jobs. Whereas these subjects are moderately experienced programmers, they have little experience with software design per se. We selected two subjects from this group (S17 and S19) and examined their thinking-aloud protocols in detail. Both these subjects had taken a course that specifically taught software design.

We also collected a protocol from a subject with no software design experience (S25, whom we call a prenovice). This subject has taken several programming courses and has written programs in the course of the research in which she is involved. Her experience differs from the novices in two ways: her formal training has dealt solely with the practical aspects of programming, and therefore she has little knowledge of the theoretical constructs of computer science; and, all her programming experience has been statistical programming in FORTRAN.

PAGE-KEYED INDEXING SYSTEM

BACKGROUND. A book publisher requires a system to produce a page-keyed index. This system will accept as input the source text of a book and produce as output a list of specified index terms and the page numbers on which each index term appears. This system is to operate in a batch mode.

DESIGN TASK. You are to design a system to produce a page-keyed index. The source file for each book to be indexed is an ASCII file residing on disk. Page numbers will be indicated on a line in the form /*NNNN, where /* are marker characters used to identify the occurrence of page numbers and NNNN is the page number.

The page number will appear after a block of text that comprises the body of the page. Normally, a page contains enough information to fill an 8½ × 11 inch page. Words are delimited by the following characters: space, period, comma, semicolon, colon, carriage-return, question mark, quote, double quote, exclamation point, and line-feed. Words at the end of a line may be hyphenated and continued on the following line, but words will not be continued across page boundaries.

A term file, containing a list of terms to be indexed, will be read from a card reader. The term file contains one term per line, where a term is 1 to 5 words long.

The system should read the source file and term file and find all occurrences of each term to be indexed. The output should contain the index terms listed alphabetically with the page numbers following each term in alphabetical order.

FIG. 8.2. The text of the page-keyed indexer problem.

The particular problem given to the subjects is to design a page-keyed indexing system. The problem specifications are shown in Fig. 8.2. This problem was chosen because it is of moderate difficulty and understandable to individuals with a wide range of knowledge of software design, but does not require knowledge of highly specialized techniques that would be outside the competence of some expert subjects; that is, a reasonable design could be constructed for this task using only the techniques taught in upper-division undergraduate courses in computer science or those contained in standard textbooks on computer science algorithms. A variety of approaches, however, could be taken to design such a system.

The protocols of a subset of the subjects were analyzed in detail, whereas others were examined more cursorily to find corroborating evidence. The method by which this analysis was carried out and the results obtained can be found in Atwood and Jeffries (1980). The discussion following is based primarily on the detailed analysis, but examples have been chosen freely from all the protocols.

Similarities Across Expertise Levels

On a first reading of these protocols, one is struck by the variations in the design solutions as much within expertise levels as across them. Both the design style of the individual subject and the set of subproblems he or she chose to attack make each solution very different from any of the others. More careful consideration, however, brings up many similarities, both within experience groups and across all the subjects.

Almost all the subjects approached the problem with the same global control strategy: Decompose the problem into subproblems. They began with an initial sketchy parse of the problem, which we have called the solution model. Some subjects were quite explicit about their solution models, whereas for others it was necessary to infer the underlying model. Whenever a subject made a quick, smooth transition from one element of the solution to the next, without any overt consideration of alternatives and without reference to external memory, we assumed that the solution model underlay this decision.

The solution models for the indexer problem are surprisingly similar for both experts and novices. In general, subjects decided to read in the terms, build some sort of data structure to contain them, compare the terms to the text, associate the page numbers with each term, and output the terms and page numbers. We do not assume that this would be true for all software design problems. The indexer problem was chosen to be "straightforward"; for such a problem, expertise is needed not for the initial solution model but for the expansion of this model into a well-defined set of subproblems and the further refinement of those subproblems. Our results are therefore potentially limited to similar straightforward problems. In tasks for which the determination of a solution model is itself a difficult task, quite different problem-solving methods may be used. Once the initial solution model was derived, the subjects attempted to expand this iteratively. No subject went directly from the solution model to a complete solution. They broke the problem into subproblems and refined the solution through several levels.

As a group, the novices explored a set of subproblems similar to those examined by the experts. The initial decomposition led to equivalent constituents, and, in further iterations, the novices as a group developed subproblems that were still comparable to the experts. The experts tended to examine more subproblems and frequently found different solutions. Even for idiosyncratic aspects of the problem, however (e.g., how to treat hyphenated words, terms that cross page boundaries), the novices were as likely as the experts to incorporate a particular element into the solution.

Although the novices applied the same general problem-solving methods as did the experts, their solutions were neither as correct nor as complete. Furthermore, the novices were not able to apply the more efficient problem-solving processes that the experts used. The novices were lacking in skills in two areas:

processes for solving subproblems, and ways of representing knowledge effectively.

Subproblem Solution Processes

Decomposition. When these subjects, both experts and novices, perceived a particular problem to be complex, they decomposed it into a collection of more manageable subproblems. The experts, of course, were more effective than the novices at doing this. They showed some stylistic differences in when and how they used the decomposition process, but its use is pervasive in all four expert protocols.

S2's protocol is an almost perfect example of solution by repeated decomposition. He is a proponent of design by stepwise refinement; in this protocol, he rigidly adheres to such a strategy. His initial decomposition is a listing of the major steps to be accomplished, little more than a precise reformulation of his solution model. On the next iteration, he adds a control structure to this collection of modules. Successive passes decompose these modules into sets of submodules until he is satisfied that he has reached the level of primitive operations.

S3 also iteratively decomposes the problem in a top-down, breadth-first, beginning-to-end manner. Her style and the design she eventually produces are similar to that of S2, except that her protocol is interspersed with digressions that relate to subproblems at other levels and at other positions in the problem. S3 also attempts fewer iterations than S2, bringing the problem to a slightly higher level of detail in two passes as S2 did in five or six. In fact, at the end of the protocol, she realizes that the second iteration is so much more detailed than the first that it taxes her ability to comprehend the solution. She then incorporates a sketchy third iteration at a "higher" level than the previous one.

After articulating his problem model, S5 notes that in order to know how to read the term file into a data structure, he needs to know more about how the matcher works. He then proceeds to work out the design of the matcher and its associated data structures. This places him directly in the middle of the decomposition tree, working simultaneously on two distinct branches. After ascertaining how the match process would operate, he proceeds to flesh out the design, proceeding from here in a top-down, breadth-first, beginning-to-end manner.

The core of S35's solution is an algorithm he retrieves that defines the term data structure and the matcher. Using this as a base, he builds the design in a top-down, breadth-first manner, although he does not expand it beginning to end. The reason for this is that he defines the problem in terms of data structures derived from his original functional analysis of the problem decomposition. Occasional deviations from this breadth-first order occur when he attempts to define low-level primitive actions that are the building blocks of his design.

All these experts demonstrate the existence of a polished design schema and a sophisticated ability to use the decomposition method to expand their designs.

Differences across experts were in part dictated by disparate design styles but to a great extent were due to differences in their knowledge of and ability to retrieve a relevant solution plan.

The novices, on the other hand, were much less effective in their use of the iterative decomposition method. They seem to lack the more subtle aspects of the design schema. A well-developed schema should guide the designer toward the production of a "good" design, as opposed to one that accomplishes the task "by hook or by crook." This means that considerations of efficiency, aesthetics, etc. should influence the manner in which design elements are expanded. There is no evidence of this in the novices. Furthermore, the schema should include procedures that enable designers to make resource decisions about the order in which to expand the modules (e.g., most difficult first, or a module that uses a data structure might be designed before the one that produces it). In the novices that we have examined in detail, we see no deviations from the default breadth-first, beginning-to-end consideration of modules.

The best of the novices was S19. He is the only novice that iterates the problem through more than two levels of decomposition. However, beyond the first level, he is unable to recursively apply some of the same decomposition strategies he used earlier. S19 gets particularly bogged down in his "compare" routine, rewriting it several times without complete success. On each attempt, he simply tries to generate a solution through brute force by writing down the necessary steps. There is no hint of having generated a model for this process nor of any attempt to further decompose it.

S17 was able to decompose the indexer problem and to generate an adequate initial pass at a solution. He then attempted to expand his solution (mostly at the urging of the experimenter). However, he makes no attempt to further decompose his chosen modules. Each subsequent iteration simply repeats the previous solution, adding on new "facts" as he discovers them. For example, at one point he considers the possibility that a term straddles pages. He changes his design to accommodate this, but he does so by augmenting existing elements, not by decomposing them into submodules. This sort of behavior indicates that S17 is unable to recursively apply the design schema.

Another of the novices writes down a solution in terms of steps, instead of modules. The distinction between steps and modules is necessarily a fuzzy one. However, a set of steps differs from a modular decomposition in that steps have no hierarchical structure, steps of very different levels of detail may occur together, and steps have only a primitive control structure. In the second iteration of his design, this novice merely produces a similar set of steps, more specifically tied to the architecture of a particular computer. He appears to understand that a problem should be broken down but has not developed a design approach that decomposes into subproblems.

Although the novices have not incorporated the more subtle aspects of the design schema into their behavior, they can apply the basic principles. The

prenovice, S25, however, has not developed even a rudimentary design schema. First, S25's protocol is qualitatively different from those of the computer science majors. They produced designs that, although differing in many details from those of the experts, were at least marginally acceptable solutions to the problem. S25 did not produce a design. She generated a mixture of FORTRAN code and comments that together could be taken as a partial solution to the task of writing a program to solve the indexer problem. Moreover, she got quite bogged down in the selection of data structures for the text and terms and in the implementation of procedures to compare items in these structures. Because of these difficulties, she eventually abandoned the task without generating a complete solution.

S25 made no attempt to decompose the problem; she did not seem to be using any kind of an overall model to guide her solution. She let the problem description and the portion of the "program" already written direct her expansion of a solution. Information did not seem to accumulate over the solution attempt; she attacked the same subproblem repeatedly but often made no progress beyond the initial attempt. She did seem to understand that input, process, and output components were needed, but this was not sufficient to produce a correct initial decomposition of the problem.

We take this continuum of more effective use of the decomposition method with increasing experience as strong evidence for both the reality and the usefulness of the design schema. Another aspect of expertise that is apparent in these protocols is the ability of the experts to generate and evaluate alternative solutions to a subproblem.

Evaluation of Alternatives. When the experts are trying to determine whether a particular plan is actually a good solution to a subproblem, they state alternative solutions and select among them. S3, for example, explicitly mentions that the page numbers could be stored in an array or a linked list. She does some calculations of the relative storage requirements of each and chooses the linked list because it is more efficient. S35 spends some time considering two ways of implementing his term data structure; one is time efficient, and the other is storage efficient. He concludes that, without knowledge of the actual computer system to be used, he does not have enough information to decide which is better. He chooses to leave both as alternatives.

The novices seldom consider more than one possible solution to any subproblem. From the marginal utility of some of the solutions they do retrieve, it seems that they are hard pressed to find even one solution to many subproblems. For example, at one point, S19 says "This might be the only way I can think of to be able to do this. It's going to be awful expensive," and elsewhere, "It's inefficient and expensive, but it's easy." He seems to have some ability to critique his solutions but is at a loss to correct the deficiencies he finds.

In the few cases in which the novices choose among alternatives, they make simple dichotomous decisions (do X or not X). Their decision is invariably made

on the basis of programming convenience. For example, S19 notices that a term could straddle a page. He spends some time deciding whether or not to permit this and decides that it is easier not to allow it, although this solution is unlikely to be realistic in terms of indexing a textbook.

Retrieval of Known Solutions. One of the features of the decomposition technique is that it enables the designer to convert a problem into a set of simpler subproblems, eventually reaching the point where all the subproblems have known solutions. Although the novices attempt to employ decomposition, we see no evidence that they do so in order to arrive at a set of known solutions. The experts, in contrast, seem to have a large repertory of solutions and of methods for decomposing a problem. The clearest examples of this are when some of the expert subjects were able to recall and apply a single solution to the major problem tasks. S35 and S5 both attempted this.

S35, after reading the specifications, immediately states "Well, the obvious answer to this is to use the technique of Aho and Corasick, which appeared in CACM (Aho & Corasick, 1975)." This article describes an algorithm for searching text for embedded strings. He says: "basically what you do is you read the term file, and you create a finite state machine from it. And then you apply this finite state machine to the text." S35 then spends the next 2 hours expanding this solution into a complete design, incorporating the idiosyncrasies of this problem (e.g., that the page number is not known until the end of the page) into this general algorithm. It is apparent that his understanding of the algorithm strongly influences the expanding design and many of the design decisions.

After his initial parsing of the problem, S5 notes that the match process is critical for an efficient and successful solution. This reminds him of a published algorithm (Boyer & Moore, 1977) that may be applicable to this situation: "Now my immediate inclination is to, about three CACMs ago, this particular problem was discussed." The algorithm he refers to is similar to the one recalled by S35.

S5's memory of this algorithm is somewhat sketchy, though, and he is unsure of how it interacts with the rest of the design. He works through the match process and its associated data structure in some detail. The resulting algorithm is similar to, but not identical with, the published algorithm. In a very real sense, he constructs an original solution that incorporates many of the features that he recalls from the Boyer and Moore algorithm. From there, he proceeds iteratively through refinements of the design as a whole.

Our other two experts, S2 and S3, did not retrieve a single solution to the major tasks, but they frequently solved subproblems by incorporating plans that they had used before. For example, S2 uses a linked list to store the page numbers. He notes that the insertion procedure is somewhat tricky to implement; he would prefer to refer to one of his earlier programs, rather than spend the time to work out the details again. S3, when considering the problem that hyphens can serve two distinct functions in the text (as part of a word or to divide a word at a line boundary), mentions that she knows of a similar case that was solved by requiring that distinct characters be used in each case.

The experts not only retrieve solution plans to all or part of the problem, but they are able to modify those solutions to fit the current situation. S35's design was a modification of a well-understood plan. S5 only retrieved the skeleton of a plan; he spent most of his time augmenting and altering this plan to fit the actual problem.

The novices show no evidence that they are trying to adapt previously learned solutions to any part of this problem. No novice ever made a statement like "this is just like X" or "I did something similar when Y." They do retrieve solutions, but only at the lowest levels. For example, S17 decided that he would flag the first empty position for each term in his page-number array. This is a solution to the problem of locating the current end of the page-number list, but it is far from the best one. S17 makes no attempt to alter this solution so that it accomplishes this in a more efficient manner. It is not clear whether this is due to his inability to realize the inefficiencies in this solution, or whether he simply does not know what modifications to make.

Knowledge Representation

Access to Background Knowledge. The experts demonstrated an impressive ability to retrieve and apply relevant information in the course of solving this problem. The appropriate facts are utilized just when they are needed; important items are seldom forgotten. Moreover, they devote little time to the consideration of extraneous information. In contrast, the novices' lack of an adequate knowledge organization for solving this problem is apparent throughout their protocols. They frequently fail to correctly apply knowledge that is needed to solve the problem, and the information that they generate in the course of solving the problem is often not available to them when it is most needed. We attribute this, in part, to the inadequacy of the organizing functions provided by their immature design schemata.

The novices' failure to apply relevant knowledge can be seen in their selection of a data structure for the terms and page numbers. Each term can potentially have a very large number of page references associated with it, but the typical entry will have only a few references. The selected data structure should allow for the occasional term with an extreme number of references without having to reserve large amounts of storage for every term. A linked list is a data structure that allows these properties. Our experts used a linked list to hold the page numbers associated with each term. The course from which the novices were recruited had recently covered linked lists. In addition, most, if not all, of them had been exposed to this concept in other courses. Thus, we are confident that the subjects were familiar with the construct. In spite of this, none used such a list to hold the page numbers. They all stored page numbers in an immense array.

Several subjects mentioned that such an arrangement was inefficient, but none were led to change it.

The construction of a linked list is a technique with which these subjects are familiar. However, their understanding of when that technique is applicable does not extend to the current situation. Understanding of the conditions under which some piece of knowledge is applicable is one way in which knowledge about a domain becomes integrated. For this information to be useful, it cannot exist as a set of isolated facts but must be related to other knowledge. For example, linked lists would be interrelated with information such as additional types of data structures and methods of gaining storage efficiency in a program. The experts have achieved this integration of concepts, although it is still undergoing development in the novices.

Episodic Retrieval. The design schema mediates retrieval of information within a problem-solving effort as well as retrieval of relevant background knowledge. The experts, with their more mature design schemata, were better able to accumulate useful information during the course of the solution attempt and to apply it at the relevant time. The clearest example of this is S3's handling of the issue of hyphens in the problem.

Early in the protocol, S3 notices that the text may contain hyphens and that this complicates the comparison process. At this point, S3 only notes this "as being a problem when you come around to comparing." This issue is not considered for long portions of the protocol, but it emerges whenever a module that is related to the compare operation or accessing the text is considered. S3 never mentions hyphens when she is expanding the "read terms" module, but it is one of the first things mentioned when the "construct index" module is taken up.

In contrast, the novices are not only less able to generate relevant information, but the information that they do generate is not stored in an easily retrievable form. S19 provides an illustration. Early in his solution, he notes that a term may straddle a page. He decides that this possibility complicates the design unnecessarily and legislates that it will not happen. He even writes down this assumption. Sometime later he again notices that this problem could occur. He treats this as an entirely new discovery; no mention is made of his earlier treatment of the topic. In fact, during this second episode, he decides to allow terms to straddle page boundaries but uses the ending page number instead of the starting page number as the reference. This too is written down, but neither then nor later does he notice that it contradicts his earlier assumption.

Another example is that S17 mistakenly assumes that terms will be single words, rather than phrases. In the middle of the problem, while rereading the specifications for some other purpose, he notices the error and comments on corrections that must be made to allow for multiword terms. However, none are incorporated into his next iteration of the problem, which only deals with single-word terms. At the end of the session, he notices once more that terms are phrases and that his design must be modified to account for that fact.

This failure to recall information over the course of a single solution attempt is probably the result of two handicaps under which the novices must operate. First, the solution to these problems consumes such a large portion of their resources that they are unable to monitor memory for other potentially relevant information. Experts can avoid overloading themselves by utilization of the design schema. Second, their memory representation of the problem is not organized in such a way as to facilitate the retrieval of previously generated information.

Understanding of Concepts. The novices fail to have an adequate understanding of many of the basic concepts of computer science. These undergraduates are generally familiar with only one machine (the CDC6400) and two or three programming languages. Much of their understanding of the basic concepts is tied to their experience with one or two exemplars of that concept and reflects the idiosyncrasies of that experience. These mistaken assumptions frequently lead to inefficient designs and occasionally to outright errors.

Several examples of incomplete or incorrect understanding of concepts can be found in the protocols of S17 and S19. S17, in particular, repeatedly attempts to incorporate constructs into his design that he is aware of but does not fully understand. He tells the experimenter that the book text should be stored as a "binary tree" [i.e., he intends to read in the book text and sort it into alphabetic order (presumably by word)]. A binary tree is an efficient structure for repeatedly searching ordered collections of items. It allows one to find an arbitrary item in the set with substantially less searching than a sequential search requires, in much the same way that one looks up an entry in a dictionary or a phone book. However, all the information as to which word follows another, which are necessary to isolate phrases from the text, is lost. S17 has apparently learned some of the conditions under which a binary tree should be used, but he clearly does not understand the concept well enough to reject it in this obviously unsuitable situation.

Contrast this with the solution of S5. He is quite concerned with efficient storage of the terms and the text. He spends over an hour working out appropriate data structures and how they will be searched, as opposed to the minute or two spent by S17. S5's solution is to store the text as a string, and, for very much the reasons mentioned previously, to store the terms in a binary tree. These decisions are exactly opposite to those arrived at by S17.

S19's protocol shows that he does not completely understand the difference between computer words and English words. On the computer that he is familiar with, a computer word will contain an English word of up to 10 characters, so for many practical purposes, the distinction is not needed. In his term data structure, he allocates five (computer) words for each term, one for each (English) word. Whereas this might not be the most effective way to store the terms, it might work for some data sets, at least on the CDC6400. His misunderstanding of the difference gets him into trouble, however, when he tries to read the text. He initially tries to read it a line at a time but abandons this because he cannot

determine how many words are on a line. He then decides to read the text a word at a time. His assumption that an English word is a natural unit for input (it is not; it takes a substantial amount of computation to determine the word's boundaries) is due to his confusion between the two types of words.

S3, on the other hand, not only understands the difference between the two concepts but is also aware that the overlapping terminology is confusing. When she is allocating list pointers, she comments "the pointers themselves are actually in a vector of NT units, or words, well, computer words, I guess, . . . (that's certainly a misused word)." Thus, she is sensitive to the distinction between the concepts as well as the confusibility in terminology.

Yet another example is S17's confusion over what a flag is and when to use one. A flag is a variable that can take on two values, usually "true" and "false." It is used to indicate the status of some condition that changes within the program. S17 has some understanding of the use of flags, as he intends to "set a flag back and forth" to signal the end of the text file. Although this is not an error, it is not a particularly good use for a flag, as the end of the text file will only be reached once, and a simple test for the condition would be more suitable.

Later on in the design, he needs a way to indicate which terms have been found on the current page before the page number is available. Although his solution incorporates the idea of setting a flag, he calls it a "count." This misuse of terminology confuses him later on, when he mistakes this "count" for a count of the number of times each term occurs in the entire text.

Understanding of Implications. In addition to their conceptual failures, the novices are often unable to extract all the implications of a piece of knowledge. In particular, they are frequently unable to derive the implications of the interactions between a task and a computer implementation of that task. This is exemplified by the differences in the way the experts and novices dealt with the subproblem that compares the text and terms.

This subproblem is the heart of this design, because the efficiency of this routine directly impacts the overall efficiency of the program. All the experts treated the matcher as a difficult problem. They concerned themselves with many aspects of it: whether comparing should be done character by character or word by word; how to organize the data to minimize the number of comparisons that are unsuccessful; what constitutes a correct match. The novices, for the most part, simply stated the subproblem and made no further effort to decompose it. They seemed to treat it as too simple to require further consideration. The experience most of the novices have with compare procedures is with those that deal with comparing numbers. For such cases, the procedure is quite straightforward. Questions about how much to compare at once and how to decide if a match has occurred never arise. The novices did not retrieve information about factors that must be considered in a character string compare, because they simply did not understand the implications of the way a computer compares data.

The novice protocols indicate that novices have mastered the jargon of the field; their comments are peppered with technical computer science terms. More careful examination, however, shows that these terms do not have the same meaning for the novices as they do for the expert. This implies that in some sense, design decisions that are described by the same words are not the "same" for people of different experience levels. In addition, as the earlier examples show, these misunderstandings and failures to deduce relevant implications frequently lead to the novices astray. They confuse similar concepts or apply a concept when it is inappropriate or do not take into account pertinent considerations. In actual designs these subtle errors could be disastrous, as they probably would not be noticed until the program was written. If the problem were serious enough that a major change to the design was required, large amounts of effort would have been wasted.

DISCUSSION

The decomposition process is central to the successful derivation of a software design. It serves to break a problem down into manageable and minimally interacting components. Thus, the task is reduced to one of solving several simpler subproblems. For experts, the decomposition and subproblem selection processes of the design schema dictate the global organization of their design behavior. They first break the problem into its major constituents, thus forming a solution model. During each iteration, subproblems from the previous cycle are further decomposed, most frequently leading to a top-down, breadth-first expansion of the solution. The iterative process continues until a solution is identified for each subproblem.

The data show a range of development in the utilization of the decomposition process. At least four distinct levels can be distinguished. The first level is exemplified by the prenovice, S25, who attempted to code the major steps of the solution directly in FORTRAN. A novice designer at the next level derives a solution model and converts it into a series of steps. Novices who broke the problem into steps were usually able to iterate over the steps at least once, producing a more detailed sequence of steps.

The more advanced novices are able to break the problem into meaningful subproblems, using their solution model as a basis. S17 is able to carry out this first level decomposition, but he is unable to recursively apply this strategy. S19 is able to recursively decompose the problem for the first few levels, but eventually he becomes so mired in details that the strategy breaks down.

The experts manifest the fourth level of development of the decomposition processes. They exhibit all three major components of the strategy: (1) They break the problem into manageable, minimally interacting parts; (2) they understand a problem before breaking it into subproblems; and (3) they retrieve a

known solution, if one exists. S2 and S3 depended almost completely on the first two of these, whereas S5 and S35 were able to retrieve a known solution to a significant portion of the problem.

Experts devote a great deal of effort to understanding a problem before attempting to break it into subproblems. They clarify constraints on the problem, derive their implications, explore potential interactions, and relate this information to real-world knowledge about the task. The novices, on the other hand, show little inclination to explore aspects of a subproblem before proposing a solution. This has serious consequences for both the correctness and efficiency of their designs.

Expert designers employ a set of processes that attempt to find a known solution to a given subproblem. Critical features of the solution model are used to search for potentially applicable algorithms. Successful retrieval requires the designer to have knowledge of relevant solutions and their applicability conditions, to be able to retrieve the solution in a possibly novel context, and to adapt the solution to the particular context of the design problem. The experts show themselves to be skilled at retrieving algorithms for use in their designs. Novices show no evidence of recognizing the applicability of information in a novel situation that they had unquestionably learned previously. The novices' schemata are deficient in the processes that control the retrieval of information for integration into their designs.

The experts differed in their ability to recall high-level solutions to the problem, specifically, for the matcher and its associated data structures. S35 retrieved an algorithm from the literature and built his solution around it. S5 retrieved a skeletal solution to the same subproblems. However, he chose to work out this solution in some detail before proceeding with the remainder of the design. S2 and S3 did not retrieve information about possible solutions to these subproblems. Instead, they used the default decomposition processes to iteratively refine the problem. Both, however, recalled numerous low-level algorithms that they incorporated into their designs.

The objective of the decomposition process is to factor a problem into weakly interacting subproblems. However, subproblems can interact, and the individual solutions must be integrated. This can impose serious coordination demands upon the problem solver (Simon, 1973). The experts used two components of the design schema to solve this coordination dilemma. First, experts expand subproblems systematically, typically top-down, breadth-first. Second, they are able to store detailed and well-integrated representations of previous problem-solving activities and retrieve them when they become relevant.

Novices have difficulty coordinating their activities because of ineffective retrieval strategies. Because they do not recognize the implications of potential interactions, novices are often unable to correctly interface subproblems. They also fail to retrieve and incorporate information acquired in the classroom and are

unable to integrate information generated during earlier parts of the solution attempt with later efforts. Thus, they do not generate a consistent and well-integrated solution to the problem.

The variations in performance, both within and between levels of expertise, demonstrate the complexities of learning the design schema. Basically, the schema is learned through actual experience in doing software designs; textbook knowledge is not sufficient. The experts' years of experience enable the procedures of the schema to become automatic, freeing the designer to focus more on the details of the specific problem. As the more sophisticated processes of the schema develop, the designer is able to deal more successfully with complex problems.

The differences in the ability to use the decomposition process demonstrate that the schema develops in stages. The levels along this continuum seem to correspond to incremental improvements in a designer's understanding and control of the decomposition process. Novices first understand that the problem has to be broken down into smaller parts, although they do not have a good understanding of the nature of those parts. Next, they add the idea that the breakdown should occur iteratively; that is, they should go through several cycles of breaking things down. At the next level, they acquire the ability to do the decomposition in terms of meaningful subproblems, and, finally, to recursively apply this strategy. The mature design schema would include at least the following additional processes: refinement of understanding, retrieval of known solutions, generation of alternatives, and critical analysis of solution components.

The processes people use to solve complex problems in their field of expertise are important to the understanding of the development of that skill. In software design, these processes appear to be specialized versions of more general methods, which are highly organized and automatic. Although these processes superficially resemble the default methods, they are so strongly tailored to the specific domain that they should be considered distinct methods in their own right. For any sufficiently complex and well-learned skill, these kinds of organizational structures would seem to be necessary. A crucial question, which remains to be addressed, is what types of skills lend themselves to the development of such structures.

ACKNOWLEDGMENTS

This research was supported by the Office of Naval Research, Personnel and Training Research Programs, Contract No. N00014-78-C-0165, NR157-414. Computer time was provided by the SUMEX-AIM Computing Facility at the Stanford University School of Medicine, which is supported by grant RR-00785 from the National Institutes of Health.

We wish to thank Dr. James Voss for his instructive comments on an earlier version of this chapter.

REFERENCES

Aho, A. V., & Corasick, M. J. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 1975, *18*, 333-340.

Atwood, M. E., & Jeffries, R. *Studies in plan construction I: Analysis of an extended protocol* (Technical Report SAI-80-028-DEN). Englewood, Colo.: Science Applications, Inc., March 1980.

Balzer, R. M. *A global view of automatic programming*. In Third International Joint Conference on Artificial Intelligence: Advance Papers of the Conference. Menlo Park, Calif.: Stanford Research Institute, 1973, 494-499.

Balzer, R. M., Goldman, N., & Wile, D. *Informality in program specifications*. In Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass., August 1977.

Barstow, D. R. *A knowledge-based system for automatic program construction*. In Fifth International Joint Conference on Artificial Intelligence: Advance Papers of the Conference, Cambridge, Mass., August 1977.

Barstow, D. R. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 1979, *12*, 73-119.

Bhaskar, R. & Simon, H. A. Problem solving in semantically rich domains: An example from engineering thermodynamics. *Cognitive Science*, 1977, *1*, 193-215.

Biermann, A. W. Approaches to automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), *Advances in computers* (Vol. 15). London: Academic Press, 1976.

Boehm, B. W. Software design and structuring. In E. Horowitz (Ed.), *Practical strategies for developing large software systems*. Reading, Mass.: Addison-Wesley, 1975.

Boyer, R. S., & Moore, J. S. A fast string searching algorithm. *Communications of the ACM*, 1977, *20*, 762-772.

Green, C. *A summary of the PSI program synthesis system*. In Fifth International Joint Conference on Artificial Intelligence. Cambridge, Mass., August, 1977.

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

Heidorn, C. E. Automatic programming through natural language dialogue: A survey. *IBM Journal of Research and Development*, 1976, *20*, 302-313.

Jackson, M. A. *Principles of program design*. New York: Academic Press, 1975.

Larkin, J. H., *Problem solving in physics* (Technical Report). Berkeley, Calif.: University of California, Department of Physics, July 1977.

Levin, S. L. *Problem selection in software design* (Technical Report No. 93). Irvine, Calif.: University of California, Department of Information and Computer Science, November 1976.

Long, W. J. *A program writer* (Technical Report No. MIT/LCS/TR-187). Cambridge, Mass.: Massachusetts Institute of Technology, Laboratory for Computer Science, November 1977.

Mark, W. S. *The reformulation model of expertise* (Technical Report No. MIT/LCS/TR-172). Cambridge, Mass.: Massachusetts Institute of Technology, Laboratory for Computer Science, December, 1976.

Myers, G. J. *Software reliability: Principles and practices*. New York: Wiley, 1975.

Sacerdoti, E. D. *A structure for plans and behavior* (Technical Note 109). Menlo Park, Calif.: Stanford Research Institute, August 1975.

Simon, H. A. Experiments with a heuristic compiler. *Journal of the ACM*, 1963, *10*, 493-506.

Simon, H. A. The heuristic compiler. In H. A. Simon & L. Siklossy (Eds.), *Representation and meaning: Experiments with information processing systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

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

Sussman, G. J. *Electrical design: A problem for artificial intelligence research*. Proceedings of the International Joint Conference on Artificial Intelligence, Cambridge, Mass., 1977, 894-900.

Warnier, J. D. *Logical construction of programs*. Leiden, Netherlands: Stenpert Kroese, 1974.

Yourdon, E., & Constantine, L. L. *Structured design*. New York: Yourdon, 1975.