Chapter 1.3

The Tasks of Programming

Nancy Pennington[1] and Beatrice Grabowski[2]

[1] *Psychology Department, Campus Box 345, University of Colorado, Boulder, Colorado 80309, USA*
[2] *Department of Anthropology, Rawles Hall, Indiana University, Bloomington, Indiana 47402, USA*

Abstract

Computer programming and other design tasks have often been characterized as a set of non-interacting subtasks. In principle, it may be possible to separate these subtasks, but in practice there are substantial interactions between them. We argue that this is a fundamental feature of programming deriving from the cognitive characteristics of the subtasks, the high uncertainty in programming environments, and the social nature of the environments in which complex software development takes place.

1 Introduction

A distinctive characteristic of computer programming derives from the variety of subtasks and types of specialized knowledge that are necessary to perform effectively. A skilled programmer must comprehend the problem to be solved by the program, design an algorithm to solve the problem, code the algorithm into a conventional

Psychology of Programming
ISBN 0-12-350772-3

programming language, test the program and make modifications in the program once it is completed. Success at these programming tasks requires knowledge of the external problem domain (e.g. statistics, finance, electronics, communications), knowledge of design strategies to develop and implement algorithms, knowledge of programming languages, knowledge of computer hardware features that affect software implementation, and knowledge of the manner in which the program will be used. In the present chapter, we provide an overview of computer programming in terms of programming subtasks, knowledge sources and the interrelations between these components.

Computer programming may be characterized 'as a whole' as a design task (Greeno and Simon, 1988). Examples of other design tasks include architecture, electrical circuit design, music composition, choreographing a dance, writing an essay or writing an instruction manual. There are several features that design tasks have in common. First, the goal of the designer is to arrange a collection of primitive elements in the design language in such a way as to achieve a particular set of goals. For computer programming, this involves piecing together a set of programming language instructions that will solve a specified problem. Secondly, two fundamental activities in design task domains are composition and comprehension. Composition is the development of a design and comprehension results in an understanding of a design. The essence of the composition task in programming is to map a description of *what* the program is to accomplish, in the language of real-world problem domains, into a detailed list of instructions to the computer designating exactly *how* to accomplish those goals in the programming language domain (Brooks, 1983). Comprehension of a program may be viewed as the reverse series of transformations from *how* to *what*. Thirdly, the composition and comprehension transformations are psychologically complex tasks because they entail multiple subtasks that draw on different knowledge domains and a variety of cognitive processes. For example, Brooks (1977) divides the programming task into subtasks of understanding the problem, method finding (planning) and coding. Other researchers include design, coding and maintenance subtasks. Multiple subtasks are also typical of other design tasks such as writing, for which planning, translating and reviewing have been suggested as distinct component processes (Hayes and Flower, 1980).

In Figure 1 we depict this characterization of design tasks in general, and of computer programming in particular. Basic programming subtasks of (1) understanding the problem, (2) design, (3) coding and (4) maintenance are shown. Basic processes of composition and comprehension are shown as processing tasks that cycle through the different subtasks. With each subtask we have associated certain mental products and knowledge domains, tentatively adopting Brooks' (1983) definition of the programming process – the serial mapping from one knowledge domain to another, beginning with the problem domain, through several intermediate knowledge domains, and ending with the programming language domain – as a useful framework within which to begin analysis.

The tidy separation of programming subtasks and representations shown in Figure 1 is, however, misleading as a description of software design and programming as it usually occurs for any moderately complicated programming project (see Chapters 3.3 and 4.2). The programming subtasks described above have multiple interconnections that make them difficult to separate in practice. This is also true of other complex design tasks such as architecture, planning, art and writing, in which con-

Programming subtasks	Basic processes	Knowledge domains	Mental representations	External representations
Understanding the problem	COMPOSITION ↓ / COMPREHENSION ↑	Domain knowledge (e.g., statistics, banking)	Situation model	Requirements document Specifications document
Design		Design strategies Programming Algorithms and methods Design language	Solution model Plan representation	Design document
Coding		Programming language Programming conventions	Program representation	Code
Maintenance		All knowledge domains Debugging, testing strategies Frequent kinds of error	All representations	All documents

Figure 1: The tasks of programming.

ceptualization, design and implementation will influence each other in all directions (Rowe, 1987).

One way in which there are interactions between programming subtasks is that programmers rarely complete one subtask before beginning the next. Rather, the process is better described as repeated alternation among subtasks. Thus the programmer may continue to work on 'understanding the problem' in alternation with design, coding, and revision (Malhotra *et al.*, 1980; Chapter 3.3). A second feature of the programming task that makes a clean decomposition into subtasks unrealistic is that design not only takes place at different *levels* of abstraction (more than one level of detail), but it involves multiple qualitatively *different* abstractions. Kant and Newell (1985) call these different 'problem spaces'; Pennington (1987b) labels the idea 'multiple abstractions'. This refers to the idea that the design may be described in terms of its functional specifications, its procedural interrelations, its structural or transformational properties. Different abstractions (or problem spaces) may be most useful during different subtasks, yet each is needed throughout the entire programming process (see also Chapters 1.1, 1.2 and 2.2). A third and related feature of programming that connects subtasks is the interaction between knowledge domains (Barstow, 1985; Kant and Newell, 1984). For example, the computational algorithms will often depend on knowledge of problem-solving heuristics in the application domain such as electronics, geometry or physics. A fourth aspect that complicates the programming picture is related to the multiple and often messy environments in which programming takes place (see Green, Chapter 1.2). Some environments allow easy interaction between subtasks, some impose stricter separations. In addition, subtasks are most often distributed over members of the project team, adding hefty doses of communications and social problems to already difficult intellectual tasks (see Chapter 4.1).

In sum, programming is a complex cognitive and social task composed of a variety of interacting subtasks and involving several kinds of specialized knowledge. We will use Figure 1, in its bare analytical form to organize the following sections that elaborate each subtask, returning throughout and at the end to further discussion of the interrelations between component tasks.

2 Understanding the problem and problem representation

Programming problems are unique in that they usually involve solving a problem in another (application) problem domain, such as mathematics, accounting, electronics or physics, in addition to solving the program design problem. For example, a computer program may control the scheduling of a set of elevators (Guindon *et al.*, 1987). Decisions about how the elevators should work are part of solving the application problem.

It is usually expected that these application decisions will be set out in a requirements document listing the client's goals, or the requirements may be established through interaction with the client. One thing that is now clear, is that requirements documents and client's statements of goals are never complete. In some cases clients may not know all of their goals at the outset (Thomas and Carroll, 1979); in other cases there are assumptions of 'common' knowledge that will be brought to bear in combination with the written problem description (Krasner *et al.*, 1987); in other cases there will simply be omissions. Thus, one critical aspect of understanding the problem to be solved is that the programmer possesses knowledge of the application problem domain (see Chapter 2.3).

Even in the blandest of problems, some domain knowledge will be required. For example, in the text indexing problem used by Jeffries *et al.* (1981), the problem description stated that the program must produce an index listing page numbers of all instances of certain terms in the text. It requires knowledge of texts to realize that there may be instances of terms in the text that are hyphenated and therefore need special treatment. Curtis and his colleagues, in an extensive field study of programming teams (Krasner *et al.*, 1987), found that one of the most critical problems facing teams designing complex systems was that programmers often lacked critical domain knowledge that would allow them to detect incompletenesses in complex requirements documents or in initial discussions of system plans. Adelson and Soloway (1985) demonstrated that inexperience in a problem domain was associated with programmer failure to develop a 'global model' of the design. Thus domain knowledge is critical for providing a context for interpreting requirements, detecting incompleteness, constraining initial design solutions, and developing a global design model. However, even *experts* in a problem domain will not interpret system requirements identically, especially when they are stated in high-level terms (Thomas and Carroll, 1979).

A second critical feature of understanding the problem is the *form* of the problem representation. By problem representation, we mean a mental conception of the problem to be solved and/or an external characterization in the form of text or diagrams. It is well documented in the psychological literature that the representation of the problem that is constructed is an important determinant of the range of solutions that will be considered and is also an important source of problem-solving difficulty (see e.g. Hayes and Simon, 1977). One frequently cited example of this is a brain teaser called the mutilated checkerboard problem in which the task is to decide whether or not a checkerboard that has certain squares removed can be completely covered by rectangles the size of two adjoining squares. Although the problem is most readily represented in the spatial terms of checkerboards and rectangles, it is most easily solved when it is represented in terms of the *numbers* of squares of each colour that remain after mutilation. Similar results have been found in studies of

design: both the surface form (e.g. spatial or temporal) of the problem and the structuring of requirements will influence problem representation and consequently the characteristics of problem solutions (Carroll *et al.*, 1980; Ratcliffe and Siddiqi, 1985).

For programming problems it is clear that understanding the problem will result in a mental conception of the problem and that there will also be external notes and documents. Questions of interest concern what form the problem representation does take or should take, and what kinds of information are explicit at this stage of analysis. One view of problem comprehension and representation stresses an external problem domain model that includes descriptions of the various objects, their properties and relations, their initial and final states, and the operations available for going from initial to final states (Brooks, 1983; Goldman *et al.*, 1977; Miller and Goldstein, 1977). This conception of the appropriate problem representation is analogous to the development of a 'situation model', a mental representation of the situation to which the to-be-solved problem refers (van Dijk and Kintsch, 1983). For example, Kintsch has found that in solving algebra word problems, a key to success is having constructed an accurate mental representation of the real-world situation described in the problem. People who did *not* do this often correctly solved the *wrong* problem (Cummins *et al.*, 1988). Similarly in programming, Pennington (1987a) has found that the best programmers constructed a mental representation of the real world problem domain. Some, but not all prescriptions for programming embody this concept. For example, the Jackson design methodology (Jackson, 1975, 1983) advocates careful description of various domain objects, their properties and relations, as a first step in designing a computer program.

It has also been suggested that expert programmers have 'problem categories'. That is, when presented with a design problem, the designer will determine its nature by associating it with previously known problems. For experts, it is thought that each of the problem categories is closely associated with a problem solution plan that applies to problems of that type, in contrast to novices who might categorize problems according to surface features such as application area. Thus, when the expert recognizes that a problem is of a certain type, he/she also knows how to begin solving it. In an empirical test of this idea, Weiser and Shertz (1983) found that novices tended to categorize problems by application area and experts classified by algorithm. Other studies of novice programmers suggest that the ability to classify simple problems according to type distinguishes the talented from the average novice (Kahney, 1983) and that the ability to retrieve abstract solution types is an important aspect of learning to program (Anderson *et al.*, 1984). Studying initial problem representation in terms of problem categories is limited because most real-world programming problems are too complex to be assigned to a single category and the study of programming problem categories is more relevant to the later design and planning processes, when smaller subproblems are identified and when design strategies form the basis of categories (Hoc, 1988).

For practical reasons, this aspect of programming (understanding and representing the programming problem) has not been much studied; researchers have even tried to choose programming problems to study for which little specialized domain knowledge is required (e.g. Jeffries *et al.*, 1981; Pennington, 1987a,b). However, there is ample evidence in other problem-solving domains and some evidence in programming research that how the problem is understood and represented is of critical

importance in how easily it is solved, and in the correctness of solution (Hayes and Simon, 1977; Larkin, 1983; Kintsch and Greeno, 1985). Secondly, since programming may be seen, in part, as successive transformations of the external problem domain representation into the programming language representation (Brooks, 1983), the relations between domains will critically determine the difficulty of the programming task.

3 Program design and the representation of programming plans

Program design plays a central role in the programming process. Design in this context is often considered to be synonymous with planning, that is, laying out at some level of abstraction, the pieces of the solution and their interrelations. Because plans are structures that are subject to inspection and reorganization, planning serves the dual functions of preventing costly mistakes and simplifying problem solving by providing an overall structure for the problem solution (Anderson, 1983). For programming, most researchers and practitioners agree that this is an important step in determining the quality of the final program, particularly for large or complex programs. In this phase, design strategies and design knowledge are co-ordinated to map the problem representation into a program plan (see Figure 1). Design occurs at both a *general* design level in which requirements and specifications are decomposed into a system structure and at a *detailed* level in which algorithms to implement different modules are selected or created (Yau and Tsai, 1986).

The design of software, as with the design of any complicated artifact, presents challenging problems for the developer. One particular source of difficulty results from the fact that the design subtask interacts with other programming subtasks. That is, as we have already noted, design will alternate with work on problem understanding, coding and revision. One reason this occurs is because of the high degree of uncertainty and incomplete information that is typical of a large-scale programming project. Uncertainty may have many causes but some examples include changing requirements or interfacing technologies under simultaneous development. Information may be incomplete when clients are not clear about their own goals, developers are inventing new products or no one has experience with the application area. A second reason for alternating among subtasks is that decisions made in later subtasks, and problems discovered in later subtasks, may alter decisions made in previous subtasks, necessitating a return to problem understanding or early design phases.

The dominant view of planning discussed in the psychology and artificial intelligence literatures is one of step-wise refinement (Sacerdoti, 1977), in which the primary process is one of top-down, breadth-first decomposition. In this method, a complex problem is decomposed into a collection of (ideally) non-overlapping subproblems. The subproblems are decomposed into further subproblems and this is repeated until the subproblems are simple enough to be solved by retrieving or specifying a known plan for solution (Wirth, 1974). There is some empirical support for software design by step-wise refinement (Jeffries *et al.*, 1981; Carroll *et al.*, 1979; Adelson and Soloway, 1988) and some automatic programming models design in this way (Miller and Goldstein, 1977). However, as a view of what the design subtask actually involves, design by step-wise refinement presents an overly simple view.

First, there is evidence to suggest that the design process is not as orderly as that required by step-wise refinement. Miller and Goldstein (1977) found that in many

instances their computer coach needed a mechanism to alter the coach's approved (orderly) expansion. Other data also suggest that there is some amount of alternation between levels of planning as early decisions have implications for later steps and later steps may call into question some aspects of earlier decompositions (Atwood and Jeffries, 1980; Ratcliffe and Siddiqi, 1985). Some are even more pessimistic, suggesting that good programmers 'leap intuitively ahead, from stepping stone to stepping stone, following a vision of the final program; and then they solidify, check, and construct a proper path' (Green 1980, p. 306).

This evidence is more consistent with a second view of planning called opportunistic planning (Hayes-Roth and Hayes-Roth, 1979) in which the plan exists at different levels of abstraction simultaneously and the planner continually alternates between levels. The characterization of planning in this view is multidirectional rather than top-down since observations that arise from planning at lower levels may guide planning at a more abstract level. Evidence from a variety of planning domains suggests that alternation between levels of planning, coping with interdependent subgoals, and opportunism in planning are frequently observed (Rowe, 1987). Opportunism in planning has been explicitly described in work by Visser (1987, 1988; Chapter 3.3) and by Guindon (Guindon *et al.*, 1987). These researchers present ample evidence that software designers skip between levels of detail in design development. It is likely that step-wise refinement methods are used when problems are familiar and of reasonable size, and that more complex and novel design problems must of necessity involve opportunism and less-balanced design.

To describe the design subtask in terms of step-wise refinement also glosses over a major design difficulty: deciding what the units of problem decomposition are or ought to be. This, of course, is the topic of most books on programming technique (see e.g. Wirth, 1974; Dahl *et al.*, 1972; Jackson, 1975, 1983; Yourdon and Constantine, 1979), and is a matter of some dispute (Bergland, 1981). If one subscribes to the original structured programming sequencing constructs, then structural decompositions, even at the abstract level, will focus on the scheduling scheme of subprocedures. Jackson (1975, 1983) argues that the process rather than the procedure is appropriate as a fundamental structural component because of its suitability as a modelling medium of the external problem domain. Yourdon and Constantine (1979) stress dataflow decomposition. There is little empirical research on what kinds of decompositions programmers produce in designing complex programs (see Ratcliffe and Siddiqi, 1985; Guindon *et al.*, 1987, for exceptions) and it is reasonable to expect that the nature of program decompositions will be related to the type of programming task, the programming language, the programmer's training and knowledge. Different decompositions, however, will have implications for later ease of comprehension and maintenance of the program (Bergland, 1981), depending on the extent to which different types of relations between design parts such as the purpose or function of a particular plan unit, the structure of data objects, the sequencing of operations (control flow), and the transformations of data objects (data flow) are explicit or obscure (see Green (1980) and Pennington (1987b) for more discussion of multiple decompositions of program plans).

In their pure forms, problem decomposition and step-wise refinement represent 'analytic' approaches to design (Carroll and Rosson, 1985) and, as descriptions, they miss some important features of design problem solving. For example, prototyping methods of design often involve implementing a kernel solution and then adding to

the design by increments (Boehm *et al.*, 1984; Ratcliffe and Siddiqi, 1985; Kant and Newell, 1984). Discovery aspects of design involve trial solutions, keeping features that work and discarding those that don't (Carroll and Rosson, 1985). Mental simulation to estimate design effects and interactions is used extensively (Adelson and Soloway, 1988). It is also the case that design often does not start from scratch, but rather, a prior design is used as a starting point and modified (Visser, 1988; Carroll and Rossen, 1985; Pennington, 1988; Silverman, 1985).

Design is also a very knowledge-intensive subtask, utilizing several different kinds of knowledge (see Figure 1). Brooks (1977) has suggested that an expert programmer may have 50 000 to 100 000 chunks of programming knowledge. The nature of these knowledge chunks would be critical in determining problem decompositions in design. Soloway and his colleagues have described this knowledge base as 'plan knowledge'; a program plan is a stereotypical sequence of program actions that accomplish a certain computational goal (Soloway *et al.*, 1988; Rich, 1981; Chapter 3.1). Other kinds of knowledge implicated in design are: knowledge of design and programming conventions and efficiencies (Soloway *et al.*, 1988), knowledge of design methods and strategies (Jeffries *et al.*, 1981; Guindon *et al.*, 1987; Vessey, 1985: Chapter 3.2), and in some cases, knowledge of a program design language (Ramsey *et al.*, 1983).

4 Coding

Coding a program involves translating the most detailed level of the plan formulation into a programming language. In spite of the importance assigned to the *design* of the program, some studies of programming have found that planning occupies a relatively small amount of design time compared to the programming phase. For example Visser (1987) found that her subject spent 1 hour planning compared to 4 weeks coding. This implies that, in practice, design is intimately intertwined with coding in contrast to a common assumption that coding doesn't begin until the design is complete. This is especially true in environments in which prototyping design methods are prevalent. Although, in principle, a prototyping method does not preclude 'requirements before design before coding', in practice the implementation of kernel solutions often precedes full understanding of the problem or full design (see Chapter 3.3). It is also true in environments where high uncertainty causes certain design decisions to be held up or to be made only tentatively and coding works around these problem areas.

The coding process has been described as one of symbolic execution, in which a plan element triggers a series of steps through which a piece of code is generated and then the programmer symbolically executes (i.e. mentally simulates) that piece of code in order to assign an effect to it. This effect is compared with the intended effect of the plan element; any discrepancy causes more code to be generated until the intended effect is achieved. Then the next plan element is retrieved and the process continues (Brooks, 1977). In coding, perhaps more than in any other subtask, the creation of segments of code alternates frequently with evaluation, not only for correctness, but for style, efficiency, consistency of notation, etc. (Gray and Anderson, 1987; Visser, 1987). This alternation between composition and evaluation leads inevitably to the reconsideration of design decisions when the implementation in code fails to achieve the intended effects or does so awkwardly.

To the extent that the plan elements are lost from memory or never completely specified, the evolving code may serve to help reconstruct or alter the guiding plan (Green *et al.*, 1987). In addition, the coding process may be interrupted and changed as a result of processes such as symbolic execution and the evaluation of programming effects. Gray (Gray and Anderson, 1987) calls these episodes 'change episodes' and estimates that they occur as frequently as once every minute, giving the coding process a sporadic and halting nature (Green *et al.*, 1987).

It is well established that when people are learning to write computer programs, they will often find an example piece of code to use as a model for the piece of code they wish to write. However, for experienced programmers, coding is most often described as if the code were generated directly from the plan elements, i.e. from scratch. However, observations of programmers coding real (as opposed to toy) programs reveals that *most* coding does not occur from scratch (Visser, 1987; Pennington, 1988). Rather, the programmer finds a program or parts of other programs to use as coding models, or blatantly borrows old code and modifies it. This means that an important part of programming will be the ability to easily access relevant examples of code that are similar to the program under construction.

Throughout the coding process a representation of the program is built up, storing information about the objects (variables, data structures, etc.), their meanings and their properties. In Brooks' theory and in other speculations about coding behaviour (Shneiderman, 1980; Barstow, 1979), syntactic knowledge is generally considered to be represented independently of semantic programming knowledge. Syntactic knowledge is thought to include a relatively small number (Brooks (1977), estimates 50 to 154) of coding templates detailing internal statement order and syntax, in contrast to the large size estimates of semantic programming knowledge. It is, however, difficult to separate programming language knowledge from the various kinds of knowledge implicated in design phases. Knowledge that is necessary in the coding subtask also includes knowledge of the system on which the program will be implemented and the constraints that that imposes on the code.

The study of coding also cannot be separated clearly from planning in that a definitional line between refinement steps leading up to coding and the actual translation into code may be somewhat arbitrary. Moreover, certain aspects of coding must be taken into account during the design phase, such as language constraints and hardware constraints. Failure to do this has resulted in the creation of whole programs that cannot be executed on the target hardware because they, for example, exceed memory restrictions, or depend on transmission speeds not supported by the device (Krasner *et al.*, 1987).

At one or more points in the development of software, the code must be systematically tested for its correctness. Testing may be performed by the person who coded the program, or may be performed by a separate person or group. It is usually impossible to explicitly test the program on all possible combinations of inputs to the program that might occur in practice, but this is the goal. The development of and knowledge about testing methods and test data generation is a software development skill quite distinct from coding or design (see Chapter 4.2). It is the difficulty of testing for program correctness, among other things, that has led some to advocate formal proofs of correctness as an essential step in program development. Unfortunately, these methods have not yet proved practicable for programs of any realistic complexity.

5 Program maintenance subtasks

Program maintenance subtasks include debugging and modification. Both of these subtasks rely heavily on program comprehension (Jeffries, 1982; Gugerty and Olson, 1986; Nanja and Cook, 1987; Littman *et al.*, 1986). Thus we will also include program comprehension as a maintenance subtask although we believe it is a more fundamental process crossing all subtasks as shown in Figure 1. Green (1980, p. 307) writes, 'Understanding is what it's all about'. In spite of this, we do not have a complete picture of what is involved in program comprehension; as a consequence, our understanding of debugging and modification is similarly limited.

5.1 Program comprehension

Understanding a program involves assigning meaning to a program text, more meaning than is literally 'there'. A programmer must understand not only what each program statement does, but also the execution sequence (control flow), the transformational effects on data objects (data flow), and the purposes of groups of statements (function) (Pennington, 1987a,b). In order to do this, the programmer will employ a comprehension strategy that co-ordinates information 'in the program text' with the programmer's knowledge about programs and the application area. This results in a mental representation of the program meaning. A variety of strategies and representations have been suggested.

It has been suggested that program comprehension involves successive recodings of groups of program statements into increasingly higher-level semantic structures (Shneiderman, 1980; Basili and Mills, 1982). Thus, patterns of operation are recognized as higher-order chunks; patterns of chunks are recognized as algorithms, and so on. This view suggests that a hierarchical internal semantic representation of the program is built during program comprehension from the bottom up. An alternate view is that comprehension is a process of hypothesis testing and successive refinement (Brooks, 1983). In this view, the meanings of the program begin to be built at the outset. For example, even with the name of the program (e.g. main file update) the programmer will have hypotheses about the general function of the program and its major constituents. This implies that the programmer accesses and activates a high-level program schema which partially guides a search for evidence about the expected program components. During the search, the programmer will generate subsidiary hypotheses until they can be matched against beacons (Brooks, 1983) which may confirm parts of the programmer's hypotheses, further refine them or suggest alternatives. Beacons may be procedure names, variable names, or stereotypical code sequences; for example, a particular manner of exchanging values in an array may be recognized as indicating a sort routine and thus serve as a beacon. The programmer's internal representation of the program starts at the top with the program's general function. As higher-level hypotheses are refined and divided, plan elements are added to the representation, followed by the integration of lower-level chunks. The comprehension of a program is complete when the lowest-level plan elements can be bound to actual code sequences in the program. Without a doubt, both top-down and bottom-up processes are involved (Pennington, 1987a,b; Curtis *et al.*, 1984; Letovsky, 1986). For example, matching program plan knowledge to code allows the programmer to make inferences about the goals of the program. This in turn leads to further predictions about program contents, a 'top-down' process. In

contrast, mental simulation of program effects is a 'bottom-up' process that enables the programmer to reason about the goal-code relations.

The mental representation of program meaning that is constructed by the programmer is thought to be multilevelled. Drawing on current theories of text comprehension, it has been described in terms of two distinct but cross-referenced representations of a text that are constructed in the course of text comprehension (van Dijk and Kintsch, 1983; Kintsch, 1986). The first representation, the *textbase*, includes a hierarchy of representations consisting of a surface memory of the text, a microstructure of interrelations among text propositions and a macrostructure that organizes the text representation. The second representation, the *situation model* is a mental model of what the text is about referentially. Because the program text is fundamentally about computations, we have called the textbase a 'program model'; similarly, the situation model refers to the application domain, and we have called it a 'domain model'. We have further proposed that the program model will be dominated by procedural relations between program parts and the domain model will be dominated by functional relations (Pennington, 1987a). Exceptional performance in program comprehension is associated with the construction of both mental representations and with mappings between them (Brooks, 1983; Pennington, 1987a).

5.2 Debugging and program modification

The term debugging is sometimes used to refer to the combination of testing and debugging, where testing is a systematic search for program errors. Here, we use debugging to refer to the activities involved in locating and repairing errors in programs once they are known to exist. These errors could have been detected either through testing or through feedback from users of the program. Much of the software in use has associated with it a list of 'outstanding bugs' or known malfunctions. Some of these bugs have consequences that are merely annoying to users of the software but others have serious economic and potentially life-threatening consequences. Thus debugging is a major activity and cost in software development and maintenance.

Debugging is a diagnostic task, similar to other diagnostic tasks such as medical diagnosis and electronic troubleshooting. That is, the program displays some 'symptoms' and the debugger must discover the 'disease' that is causing those symptoms, and 'treat' the disease until the program is 'well' or symptom free. As such, program diagnosis (debugging) usually involves the following activities: (a) understanding the program or system of programs being debugged; (b) generating and evaluating hypotheses concerning the problem; (c) repairing the problem; and (d) testing the system, once repaired (Clancey, 1988; Katz and Anderson, 1988; Vessey, 1985, 1986).

Debugging largely involves understanding what a program *is* doing and what it is *supposed* to be doing (Kessler and Anderson, 1986; Gugerty and Olson, 1986; Nanja and Cook, 1987), and experts in debugging spend much longer understanding the program than do novices (Jeffries, 1982). As a consequence, expert programmers appear to develop a fairly complete mental representation (model) of the program and to understand the possibilities for program errors as causal models of error in this context (Vessey, 1985, 1986, 1989; Jeffries, 1982; Gugerty and Olson, 1986). One way to insure adequate comprehension for debugging is to have written the program yourself. However, most debugging will involve diagnosing programs written and/or designed by others. Thus comprehension skill is central to debugging.

A general debugging strategy is to form a hypothesis about what kind of a bug may create the observed symptoms, to search the code for the location of the bug, modify the program, and run the program to see what effect the changes had. A second key skill then is the ability to generate hypotheses about bugs. Some strategies observed in programmers include: using clues in the output, using tests of internal program states, recalling prior bugs that generated symptoms like the current ones, simulation of program parts, and trial and error (Gugerty and Olson, 1986; Gould, 1975).

A more general model of diagnosis has been put forth by Clancey (1988) in the domain of medical diagnosis that may have a clear application to understanding program debugging. One emphasis of this model is to be specific about the kinds of knowledge that the experienced diagnostician might possess, and how these different kinds of knowledge work together to narrow the diagnostic hypotheses under consideration. One important kind of knowledge is knowledge of diagnostic strategies such as elaborating symptoms when symptoms are known to have many possible causes, and such as thinking at an intermediate level of generality in terms of categories of errors rather than in terms of very specific errors (see also Vessey (1985); and Chapter 3.2 on the importance of strategic knowledge). Strategic knowledge, however, must work on a large network of domain-specific relations between diseases (or categories of diseases) and symptoms (or categories of symptoms). Thus the program debugger should possess debugging strategies and a network of knowledge about which kinds of errors and programmer actions result in which kinds of program bugs.

A second key idea pursued in Clancey's (1988) model of diagnosis and elsewhere in work on explanation-based decision making (Pennington and Hastie, 1988), is that diagnosis for complex problems is often not a result of simple associations between symptoms and diseases. Rather, disease (diagnostic category) is understood as the result of a causal process that in turn resulted in the symptoms (evidence). Under this view, the task of the diagnostician is to build an account of a disease process that can account for the symptoms. Thus, a program error would be understood as something that resulted from programmer or designer activity and diagnosis would centre on trying to understand what actions could have produced an error that produced the symptoms. These aspects of debugging need to be better understood.

The greatest difficulty in debugging appears to be diagnosis. Repair of the program presents many fewer problems (Katz and Anderson, 1988), at least in simple programs. In more complex programs, repairs may involve redesign and may have unanticipated consequences due to interactions between program parts or modules.

Program modification may, of course, take place outside of the context of debugging. For example, users may change their minds about what they want the program to do, resulting in added functions. Once again, it has been proposed that the basis of program modification is program understanding (Fjeldstad and Hamlen, 1983; Littman *et al.*, 1986). Furthermore, the style of comprehension has been shown to be strongly related to success in program modification. For example, Littman *et al.* found that programmers using a systematic comprehension strategy rather than an 'as-needed' strategy were successful at performing a required modification. This result was attributed to the completeness of the mental representation that resulted when a systematic comprehension strategy was used. Obviously, program modification will include, in addition to a base of comprehension, most other programming

subtasks as well. Extensive modifications will involve changes to design, extensive coding, retesting, and debugging.

6 Interrelations between programming subtasks

The subtasks of programming are not very different from the subtasks of any design task, or of any problem-solving activity that involves planning: the problem must be understood, the solution sketched out at some level of detail, the solution implemented, corrected and/or modified. Strategies of decomposition, pattern recognition, mental simulation, analogy and causal reasoning will be used. However, as we have stressed, the concept of programming as an unbroken progression through subtasks is not descriptive of programming practices for complicated programming projects.

At the level of individual cognition, it is simply difficult to delineate precisely where one cognitive activity finishes and another begins, such as the distinctions between understanding and design, between design and coding, and so on. This is partly due to our lack of knowledge of the exact cognitive mechanisms underlying these activities but it is also due to their interdependence in reality. The activity of design causes the developer to elaborate his or her understanding. The activity of coding may force one to consider the impossibility or inadequacy of design elements.

There is also inevitable programming subtask interaction at the level of the environments in which software development takes place. Software development is usually distributed over one or more teams (see Chapter 4.1) causing problems in communication and shared understanding. Within a single group, shared conceptualizations of the problem, the design, the code, and so on must be built up. External representations in the form of documents (see Figure 1) serve as central recordings of this. These documents, however, are always incomplete in that they assume certain shared knowledge. They are also notoriously slow to change to reflect current thinking, leaving informal communication as the avenue of building shared views. This problem is exacerbated when different groups are assigned different subtasks. In addition to the problem of building a concensus on the problem being solved, this conceptualization must be passed on to the group attacking the next subtask. For example, environments in which requirements are developed by one group and passed on to another group for high-level design need to pass not only an official document but also their often considerable knowledge of the problem domain. Curtis and Walz (Chapter 4.1) point to the need for team members in these instances whose knowledge 'spans the boundaries' of the two subtasks, to alleviate the information transmission problems.

Finally, software development that is not 'routine', i.e. a slight modification of something that has been done before, is usually characterized by high uncertainty and the need for tentative decisions at every phase of development. This element of the environment is pervasive and of great influence, yet people seem to treat it each time as if it is an aberration and not 'typical' (Krasner *et al.*, 1987). It is a fact of the world that people change their minds about what they want, especially when they see it or use it, and software has already come to be seen as evolving in this sense. However, it is also not uncommon for a software project to require, for one of its components, a piece of software or hardware that is being developed concurrently within the same company or even by another company or government agency. Sometimes features of this import are unknown, or change radically in the course of development, causing reverberating alterations in the development of anything connected to it. Software

development and the tasks of programming must be viewed above all within this context of high uncertainty and incomplete knowledge.

References

Adelson, B. and Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering,* **11**, 1351-1360.

Adelson, B. and Soloway, E. (1988). A model of software design. *In* M. Chi, R. Glaser and M. Farr (Eds), *The Nature of Expertise.* Hillsdale, NJ: Erlbaum, pp. 185-208.

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R., Farrell, R. and Sauers, R. (1984). Learning to program in LISP. *Cognitive Science,* **8**, 87-129.

Atwood, M. E. and Jeffries, R. (1980). *Studies in plan construction I: Analysis of an Extended Protocol,* Technical Report SAI-80-028-DEN, Englewood, Colorado: Science Applications.

Barstow, D. R. (1979). *Knowledge-based Program Construction.* New York: North Holland.

Barstow, D. R. (1985). Domain-specific automatic programming. *IEEE Transactions on Software Engineering,* **11**, 1321-1336.

Basili, V. R. and Mills, H. D. (1982). Understanding and documenting programs. *IEEE Transactions on Software Engineering,* **8**, 270-283.

Bergland, G. D. (1981). A guided tour of program design methodologies. *Computer,* **October**, 18-37.

Boehm, B. W., Gray, T. E. and Seewaldt, T. (1984). Prototyping versus specifying: a multiproject experiment. *IEEE Transactions on Software Engineering,* **10**, 290-302.

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

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies,* **18**, 543-554.

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

Carroll, J. M., Thomas, J. C. and Malholtra, A. Clinical-experimental analysis of design problem solving. (1979). *Design Studies,* **1**, 84-92.

Carroll, J. M., Thomas, J. C. and Malholtra, A. (1980). Presentation and representation in design problem solving. *British Journal of Psychology,* **71**, 143-153.

Clancey, W. J. (1988). Acquiring, representing, and evaluating a competence model of diagnostic strategy. *In* M. Chi, R. Glaser and M. Farr (Eds), Hillsdale, NJ: Erlbaum.

Cummins, D. D., Kintsch, W., Reusser, K. and Weimer, R. (1988). The role of understanding in solving word problems. *Cognitive Psychology,* **20**, 405-438.

Curtis, Forman, Brooks, Soloway and Ehrlich. (1984). Psychological perspectives for software science. *Information Processing and Management,* **20,** 81-96.

Dahl, O. J., Dijkstra, E. and Hoare, C. A. R. (1972). *Structured Programming.* London: Academic Press.

Fjeldstad, R. K. and Hamlen, W. T. (1983). Application program maintenance study – Report to our respondents. *In* G. Parikh and N. Zvegintzov (Eds), *Tutorial on Software Maintenance.* Silver Spring, Maryland: IEEE Computer Society Press.

Goldman, N., Balzer, R. and Wile, D. (1977). The use of a domain model in understanding informal process descriptions. *Proceedings of the Fifth Joint Conference on Artificial Intelligence.*

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies,* **7,** 151-182.

Gray, W. D. and Anderson, J. R. (1987). Change-episodes in coding: When and how do programmers change their code? *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Green, T. R. G. (1980). Programming as a cognitive activity. *In* H. T. Smith and T. R. G. Green (Eds), *Human Interaction with Computers.* New York: Academic Press.

Green, T. R. G., Bellamy, R. K. E. and Parker, M. (1987) Parsing and gnisrap*: A model of device use. *In* G. M. Olson, S. Sheppard, and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Greeno, J. G. and Simon, H. A. (1988). Problem solving and reasoning. *In* R. C. Atkinson, R. J. Herrnstein, G. Lindzey and R. D. Luce (Eds), *Stevens Handbook of Experimental Psychology,* vol. 2, pp. 589-672.

Gugerty, L. and Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

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

Hayes, J. R. and Flower, L. S. (1980). Identifying the organization of writing processes. *In* L. W. Gregg and E. R. Steinberg (Eds), *Cognitive Processes in Writing.* Hillsdale, NJ: Erlbaum.

Hayes, J. R. and Simon, H. A. (1977). Psychological differences among problem isomorphs. *In* N. J. Castellan, D. B. Pisoni and G. R. Potts (Eds), *Cognitive Theory,* vol. 2. Hillsdale, NJ: Erlbaum.

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

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

Jackson, M. A. (1975). *Principles of Program Design.* London: Academic Press.

Jackson, M. A. (1983). *System Development.* New York: Prentice-Hall.

Jeffries, R. (1982). *A comparison of the debugging behavior of expert and novice programmers.* Paper presented at the 1982 meetings of the American Educational Research Association.

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

Kahney, H. (1983). Problem solving by novice programmers. *In* T. R. G. Green, S. J. Payne, and G. C. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press.

Kant, E. and Newell, A. (1984). Problem solving techniques for the design of algorithms. *Information Processing and Management,* **28,** 97-118.

Katz, I. R. and Anderson, J. R. (1988). Debugging: an analysis of bug-location strategies. *Human-Computer Interaction,* **3,** 351-399.

Kessler, C. M. and Anderson, J. R. (1986). A model of novice debugging in LISP. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Kintsch, W. (1986). Learning from text. *Cognition and Instruction,* **3,** 87-108.

Kintsch, W. and Greeno, J. G. (1985). Understanding and solving word arithmetic problems. *Psychological Review,* **92,** 109-129.

Krasner, H., Curtis, B. and Iscoe, N. (1987). Communication breakdowns and boundary spanning activities on large programming projects. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Larkin, J. H. (1983). The role of problem representation in physics. *In* Gentner and A. L. Stevens (Eds), *Mental Models.* Hillsdale, NJ: Erlbaum.

Letovsky, S. (1986). Cognitive processes in program comprehension. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Littman, D. C., Pinto, J., Letovsky, S. and Soloway, E. (1986). Mental models and software maintenance. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Malhotra, A., Thomas, J. C., Carroll, J. M. and Miller, L. A. (1980). Cognitive processes in design. *International Journal of Man-Machine Studies,* **12,** 119-140.

Miller, M. L. and Goldstein, I. P. (1977). Structured planning and debugging. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.* Cambridge, MA.

Nanja, M. and Cook, C. R. (1987). An analysis of the on-line debugging process. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Pennington, N. (1987a). Comprehension strategies in programming. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology,* **19,** 295-341.

Pennington, N. (1988). *Design-by-example: a case study.* Unpublished manuscript.

Pennington, N. and Hastie, R. (1988). Explanation-based decision making: effects of memory structure on judgment. *Journal of Experimental Psychology: Learning, Memory, and Cognition,* **14,** 521-533.

Ramsey, H. R., Atwood, M. E. and van Doren, J. R. (1983). Flowcharts versus program design languages: An experimental comparison. *Communications of the ACM,* **26** 445-449.

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

Rich, C. (1981). *Inspection methods in programming.* Technical Report No. 604, MIT AI Laboratory.

Rowe, P. G. (1987). *Design Thinking.* Cambridge, MA: MIT Press

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior.* New York: Elsevier.

Shneiderman, B. (1980). *Software Psychology.* Cambridge, MA: Winthrop Publishers.

Silverman, B. G. (1985). The use of analogs in the innovation process: A software engineering protocol analysis. *IEEE Transactions on Systems, Man, and Cybernetics,* **15,** 30-44.

Soloway, E., Adelson, B. and Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. *In* M. Chi, R. Glaser and M. Farr (Eds), *The Nature of Expertise,* Hillsdale, NJ: Erlbaum, pp. 129-152.

Thomas, J. C. and Carroll, J. M. (1979). The psychological study of design. *Design Studies,* **1,** 5-11.

van Dijk, T. A. and Kintsch W. (1983). *Strategies of Discourse Comprehension.* New York: Academic Press.

Vessey, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies,* **23,** 459-494.

Vessey, I. (1986). Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics,* **16,** 621-637.

Vessey, I. (1989). Toward a theory of computer program bugs: an empirical test. *International Journal of Man-Machine Studies,* **30,** 23-46.

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

Visser, W. (1988). *Giving up a Hierarchical Plan in a Design Activity.* Research Report No. 814, INRIA, France.

Weiser, M. and Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies,* **19,** 391-398.

Wirth, N. (1974). On the composition of well-structured programs. *Computing Surveys,* **6,** 247-259.

Yau, S. S. and Tsai, J. J. P. (1986). A survey of software design techniques. *IEEE Transactions on Software Engineering,* **12,** 713-721.

Yourdon, E. and Constantine, L. L. (1979). *Structured Design.* Englewood Cliffs, NJ: Prentice-Hall.