Int. J. Man-Machine Studies (1989) **30**, 171–191

Cognitive issues in the process of software development: review and reappraisal

RICHARD J. KOUBEK,† GAVRIEL SALVENDY,‡ HUBERT E. DUNSMORE‡ AND WILLIAM K. LEBOLD‡

† *Department of Biomedical Engineering, Wright State University, Dayton, OH 45435, USA*

‡ *School of Industrial Engineering, Purdue University, West Lafayette, IN 47907, USA*

(Received 9 December 1986 and in revised form 2 March 1988)

The current information age has brought about radical changes in workforce requirements just as did the industrial revolution of the 1800's. With the presence of new technology, jobs are requiring less manual effort and becoming more cognitive-oriented. With this shift, new techniques in job design and task analysis are required. One area which will greatly benefit from effective task analysis procedures is software development. This paper attempts to lay a groundwork for developing such procedures by discussing important methodological issues, and examining current theories and research findings for their potential to identify the cognitive tasks of computer programming. Based on the review, this paper suggests guidelines for development of a methodology suitable for knowledge elicitation of the programming process.

1. Introduction

1.1. BACKGROUND AND SIGNIFICANCE

The advent of the industrial age, brought about by advancing technology, required workers to shift from craftsman type jobs to standardized manual tasks. Assisting this transition in the workplace were new procedures designed to maximize the relationship between job requirements and employee skills and abilities. One such technique, the task analysis, provided systematic collection and analysis of job related data.

Similarly, advances in technology are moving the present society into a new era, the information age, and workers are again being required to assume new roles. This time the emphasis has shifted from standardized manual tasks to cognitive-oriented tasks. Computer programming is a typical example. However, the old job and task analysis techniques (McCormick, Jeanneret & Mecham, 1969) are inadequate for these new tasks. Methodologies which can analyse the cognitive tasks of a job are required.

The field of software development, with its rich cognitive domain, is an area which could yield great benefits from the use of such a cognitive task analysis (Kessel, 1984). (A cognitive task is defined as an operation which brings the current state closer to the desired state.) These benefits include selecting employees with the

This work was performed while the first author was at the School of Industrial Engineering, Purdue University.

0020-7373/89/020193 + 19$03.00/0

cognitive skills best suited for the task and streamlining training programs. Also, programming languages can be developed and evaluated based on the inherent strengths and limitations of the cognitive skills and abilities identified.

Other benefits include an increased understanding of the programming process. Knowing this process, systems can be built to automate portions of the programming task. While automation for manual tasks has provided the basis of the industrial revolution, scientists are just now attempting to automate cognitive tasks via such techniques as artificial intelligence and expert systems. However, a current difficulty concerns acquiring information on how the human performs cognitive-oriented tasks; an area which the current paper attempts to address. If such a task analysis can be developed for the discipline of computer programming, then it seems plausible that the methodology can be transferred to similar fields, such as knowledge elicitation.

Although the possibility of developing such a task analysis is not certain, the potential payoff makes an attempt worthwhile. The present paper attempts to lay the groundwork for developing such an analysis. This is accomplished by first exploring current models and the cognitive processes they have identified. Also, the methodology used to develop each model is examined for a potential contribution to developing a cognitive task analysis procedure. Second, cognitive correlates which identify good programmers, such as mathematical ability and problem solving ability, are identified. These correlates may suggest important programming skills and the tasks which require these skills. Third, an attempt is made to identify the learning strategies and the subtasks in programming and discuss the current findings as they relate to the cognitive requirements of programming. Fourth, specific methodologies are reviewed for their effectiveness in analyzing such a task as computer programming.

Figure 1 displays the interrelationships of each component discussed in this section and the direction of their contribution to developing and performing a task analysis of programming.

2. Models of cognitive processes of computer programming

Since the goal is to lay a foundation for the development of a task analysis for computer programming, a logical starting point is to examine the literature for models of programming tasks. This can yield at least two results. First, the initial work of identifying the basic cognitive components may have already been started when empirical evidence for these models is derived. If the basic components are identified, then a methodology can be developed to focus on expanding the information base provided by these already-identified components. Second, the procedure used in developing and evaluating these models may provide insight into developing a cognitive task analysis methodology for the more detailed sub-tasks.

2.1. THE SYNTACTIC/SEMANTIC MODEL

A general cognitive model of programmer behavior is Schneiderman's (1980) Syntactic/Semantic model. Within the framework of current information processing theories, Shneiderman integrates knowledge particular to programming and identifies ways in which subtasks of programming may be processed. This model suggests

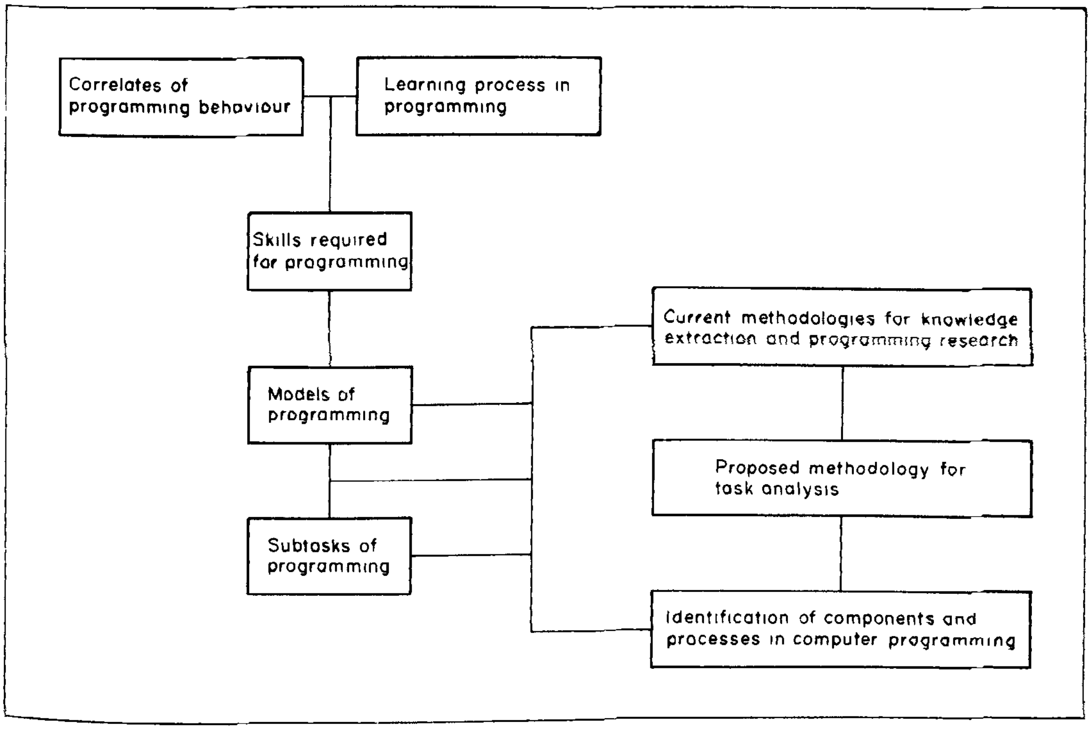
Correlates of programming behaviour
Learning process in programming
Skills required for programming
Models of programming
Subtasks of programming
Current methodologies for knowledge extraction and programming research
Proposed methodology for task analysis
Identification of components and processes in computer programming

FIG. 1. Structure of task analysis development methodology.

that the three memory systems, short term memory (STM), long term memory (LTM) and working memory, all interact during the programming process. Short term memory, a limited capacity store, is responsible for receiving input from the outside world. The long term memory contains permanent knowledge about programming and has unlimited storage capacity. The working memory has a storage capacity between the STM and the LTM and allows integration of information from the other two memory systems. During problem solving, the information from LTM and STM interact with working memory, new information is stored in the LTM, and action is taken toward the goal.

The particular areas of interest in the LTM of a programmer are the syntactic and semantic knowledge bases. Syntactic knowledge consists of detailed facts about a language, such as allowable syntax of assignment statements and looping constructs. The model suggests that this knowledge is arranged according to a particular programming language, with similar languages overlapping, and is used to express semantic knowledge in a programming language.

Semantic knowledge represents programming concepts which are independent of a particular language. The model states that semantic knowledge is arranged in a hierarchical structure ranging from low level concepts, such as function of assignment statements, to high level concepts, such as statistical analysis techniques.

In a task such as program comprehension, according to this model, the programmer first deals with the syntactical aspects of the program, such as identifying variable names, procedures, comments, and statement types. In the second stage, the lower-level semantic structure is constructed by identifying statement functions, logical chunks of code and "distinctions between procedural and declarative information" (Weiser & Shneiderman, 1986). The final two stages consist of recognizing code, such as familiar algorithms and array initialization, and identifying the relationship between the code and its application to the domain. This sequence of events represents a bottom-up approach to software development. At the very least, this model provides a broad basis for understanding programmer behavior, particularly comprehension, and for exploring the programming process in greater depth.

2.2. THE MODEL OF PROGRAM DESIGN

The Model of Program Design, which focuses on software design rather than the overall programming task, was proposed by Adelson & Soloway (1984). This model of expert behavior is based on three main properties. First, designing a program can be characterized by goals and operators. Second, the goals and operators are based on the general task of design rather than the specific task associated with the problem statement, and third, the operators interact with a knowledge base.

This model has three components. First is the Sketchy Model, which is a working representation of the program to be designed. This is revised and appended throughout the design process. The second component is the knowledge base, usually obtained from experience. This contains the information needed to solve the particular programming problem. Finally, the third and most developed component is the goals and operators.

While both Schneiderman's model and the Model of Program Design include a "refining" algorithm, the former proceeds from specific to general concepts (bottom up) while the latter moves from general to specific (top down). Also, Adelson & Soloway use a verbal report-type methodology, in which subjects speak aloud, suggesting a means for uncovering cognitive processes a programmer employs during a programming subtask.

2.2. THE CODING/COMPREHENSION MODEL

A third model of cognitive processes in computer programming was described by Brooks (1977; 1983). Part of his model relating to coding was presented in 1977 and a portion relating to comprehension in 1983. Brooks suggests three distinct states of programmer behaviour: understanding, method finding, and coding. An understanding of the problem is the first step toward a solution of a programming task. The next step is to develop a method which produces a solution set to the problem requirements, and third, this method is converted into a programming language via coding.

It was this coding procedure upon which Brooks' original model concentrated. Like Shneiderman, the memory system is a key component. A set of production rules, which consists of a set of conditions and actions to be performed when the conditions are met, are stored in the Long Term Memory (LTM). Also, information on various data structures is stored in the LTM. The STM contains new elements introduced by outside sources. According to Brooks, it is the production rules which are actually used to convert the solution method into programming code. Based on this experimental study, he has estimated that tens of thousands to hundreds of thousands of production rules are required to represent the knowledge of an experienced programmer (Brooks, 1977).

This model was later expanded to include the subtask of program comprehension. In characterizing program comprehension tasks, it is assumed that the subject is asked to identify program functions in varying depths, depending on the specific task. According to Brooks' theory, the first step in comprehending a program is to develop a primary hypothesis. This original hypothesis need not be detailed or completely accurate. It merely serves to reduce the problem space to a manageable level. Next, an attempt is made to verify the hypothesis by examining the programming code. Most likely, the original hypothesis is too broad to be verified.

Therefore, an iterative process of creating subsidiary hypotheses is continued until a depth is reached which can be verified (top down problem decomposition). The hypothesis is then verified by matching it to a section of code, or a "beacon". This beacon binds the subsidiary hypothesis with the preceding hypothesis. If a hypothesis fails, it can be a result of (1) incorrect hypothesis, (2) misuse or misunderstanding of a beacon function or (3) some unexplained lines of code whose function eludes the programmer. While programmers should consider these three possibilities in detecting errors, most time is spent on repeated attempts to understand known lines of code. By proposing these higher level tasks, this model appears to lay a foundation for a more detailed task analysis.

As did Adelson & Soloway, Brooks used verbal reports, subject notes and video taping to collect data. So far, this methodology has been able to analyse at least the higher level cognitive tasks in the programming process. Admittedly however, interpreting these protocols is rather subjective (Adelson & Soloway, 1984), their completeness may be questioned and the usefulness of verbal reports for subjects who do not verbalize well is limited.

2.4. THE MODEL OF DEBUGGING

As with coding, an existing model has already provided a list of major tasks in debugging and laid the groundwork for a more detailed follow-up. The model by Gould (1975) and Gould & Drongowski (1974) was developed to account for observed debugging behaviour, and begins with a subject's "tactic". These tactics are previously learned debugging methods used to search for a clue to the bug. Every tactic either leads to a clue or a dead end. If a clue is found, a hypothesis is formulated and a new tactic is formed based on this hypothesis. This recursive procedure continues until the bug is found. If the path leads to a dead end, the result is not necessarily void of information since the lack of a clue may also lead to a new hypothesis. This top down approach appears functionally similar to Adelson and Soloway's recursive model of software design and Brooks' 1977 hypothesis generation model of composition.

2.5. THE GOMS MODEL

A model which applies a theory of humans as symbolic processors to tasks which require human-computer interaction is the GOMS proposed by Card, Moran & Newell (1980; 1983). While this model does not attempt to identify programming subtasks, it does suggest a generic methodology for such a purpose and is therefore relevant here.

This model consists of four structures: Goals, Operators, Methods and Selection Rules (thus, the acronym GOMS). "A goal is a symbolic structure that defines a state of affairs to be achieved and determines a set of possible methods by which it may be accomplished," (Card *et al.*, 1980). Operators consist of motor and information processing acts which must be used to reach the desired goal state. A Method, in the GOMS framework, is defined as a previously-learned procedure recalled for accomplishing the task at hand. Methods are usually composed of a sequence of Operators. The final component of the GOMS model is Selection Rules which are used to select between competing Methods capable of accomplishing the same Goal. Selection Rules are in the form of If-Then statements and it is these rules which allow behaviour prediction.

The GOMS model has been used to predict time and behaviour in domain tasks such as text editing. In such limited domains, when the methods and selection rules can be enumerated, the GOMS model has great utility. The Goals and Methods naturally lend themselves to the development of a task analysis.

It currently appears difficult however to predict behaviour when the Methods, Operators and Selection Rules are either unidentifiable or very large in number. Yet, it goes far in suggesting what information should be examined in a cognitive task analysis and, in particular, how it should be structured once obtained.

2.6. EVALUATION OF THE MODELS

Upon review of these models, it appears Shneiderman's model is the most general, with Brooks', Gould's, and Adelson and Soloway's models becoming increasingly more descriptive. As Shneiderman points out, his theory provides a place to start but requires expansion and verification.

From another approach, the GOMS model is particularly well suited for suggesting which information is needed and how to organize this information. The model by Adelson & Soloway provides a fairly detailed list of tasks and procedures used in software design by experts. This appears to be a good combination of both higher level goals and detailed task description for deriving a task analysis of programming.

Brooks' and Gould's models also provide background for developing a more detailed list of cognitive tasks in programming. By dealing with broad components, they provide a framework with the flexibility to account for variance in procedures among experts. If a more detailed list of the cognitive tasks could be derived, while still maintaining the original flexibility, these models may be very useful in identifying the tasks of computer programming (Table 1 summarizes research in this area).

Based on the above review, it appears that, for many of the programming tasks, a higher level description has already been performed. These task descriptions, usually in the form of a model, have attempted to identify common processes (excluding Brooks) and therefore are very general. However, the purposes of the task analysis outlined earlier require analysis at a more detailed level. Rather than exclude individual differences, their inclusion provides an important component for

TABLE 1
Theories of cognitive processes of computer programmers

Method	Reference	Phenomenon Measured	Cognitive Model Assumed	Strengths	Possible Enhancements
Observation	Card, Moran & Newell (1980; 1983)	Text editing/ routine cognitive tasks	GOMS (Goals, Operators, Method, Selection Rules) Model	80% prediction accuracy for motor tasks. 33% prediction accuracy for time to complete task Method easily applied to routine cognitive tasks.	Account for errors. Expand applicability to routine cognitive tasks.
Protocol analysis	Brooks (1977)	Coding	Interaction of production rules	Identifies source of programmer performance differences. Flexibility to account for performance of programmers with varying levels of experience.	Include ability to make predictions of programmer performance.
None suggested	Brooks (1983)	Program comprehension	Generation and verification of subsidiary hypotheses.	Identifies source of programmer performance differences. Account for variation in problem difficulty.	Provide verification research. Include ability to predict programmer performance.
Protocol analysis	Adelson & Soloway (1984)	Program design	Goals and operators interacting with knowledge base to modify working model of the solution.	Structure of Model is domain independent. Provides a task description.	Account for errors or variance in programmer performance.
None suggested	Shneiderman (1980)	Composition, Comprehension, Debugging, Modification	Syntactic/Semantic Model	Provides a working framework upon which to build.	Provide verification research. Include ability to predict programmer tasks and performance.
Observation	Gould (1975)	Debugging	Hypothesis generation	Identifies higher level tasks in debugging.	Account for individual differences.

the purposes outlined and therefore a proposed methodology must identify them. The need for a more detailed cognitive task list, which allows individual differences, is evident.

3. Cognitive correlates of programmer performance

While these models have provided a basis for exploring the cognitive tasks in programming, other researchers have approached the question of identifying cognitive tasks from a different perspective. If the correlates of programming skill can be found, these correlates can then be used to identify the cognitive tasks, or at least characterize them. Also, the methodologies used to identify these tasks may provide a basis for developing a methodology to uncover more components of the programming task.

Probably the most documented difference between expert and novice programmers is the cognitive structure in which programming knowledge is comprehended and stored (McKeithen, Reitman, Rueter & Hirtle, 1981; Barfield, Koubek & Hwang, 1985). Studies in areas other than computer programming (Chase & Simon, 1973) have observed experts storing and recalling information in chunks whereas novices concentrate on individual items.

A chunk is composed of unitary elements of a problem grouped together based on their functional relationships. Chunking information allows the expert to consider pieces of information as a single unit and reduces the load on working memory.

Anderson, Greeno, Kline & Neves (1981) suggest a mechanism of composition and proceduralization which may account for expert chunking behavior of programming procedures. Proceduralization is the process whereby individual steps in a process are combined to form a smooth execution of the combined steps via a single activation. As the procedures develop, they are combined to form larger procedures through composition. This mechanism provides a plausible explanation for the problems experts have in recalling specific steps in their solution process.

Researchers have also identified other differences in memory representations of programming knowledge between experts and novices (Table 2). Adelson (1984) has found that, when comprehending a program, experts tend to form abstract representations containing general knowledge about what a program does. Novices, however, form a more concrete representation of how a program functions. Kahney (1983) has found similar results when allowing experts and novices to recall a program. Experts first recalled the overall structure and then the details, while novices immediately began recalling lines of code.

The description of the knowledge representations, while not directly identifying the cognitive tasks, may give clues to the identity of these tasks as further research is performed. An important point derived from this information is that novices and experts mentally represent the information differently. This suggests that the cognitive tasks which experts and novices perform in programming may be different. Also, with this information, one is able to identify factors which differentiate between expert and novice programmers.

Other correlates of programming ability which may give some insight into the cognitive tasks have also been explored. For example, numerous studies have shown that performance in mathematics, chemistry and physics courses correlate strongly

TABLE 2
Correlates of programmer performance relevant to identification of the cognitive tasks in programming

Variable studied	Reference	Method utilized to achieve results	Subjects		Results and correlations achieved with computer programming performance
			Number	Type	
Chunking	McKeithen *et al.* (1981)	Memorization and recall.	56	Beginner, intermediate and expert programmers naive, novice, intermediate and expert programmers	Experts recall more information than non-experts due to grouping information into meaningful chunks.
	Barfield *et al.* (1985)	Memorization and recall.	221		
Mental representation of programs: abstract *vs.* concrete	Adelson (1984)	Memorization and recall.	36	Novice and expert programmers. Novice and expert programmers.	Experts form abstract mental representation of programs while novices form concrete mental representations.
	Kahney (1983)	Memorization and recall.	*		
Performance in college mathematics course	Capstick *et al.* (1975)	Correlation study	46	Students with 1 year computer experience. Beginning Engineering Students.	$r = 0{\cdot}57$ with performance is programming course. $r =$ with performance in programming course.
	Barfield *et al.* (1983)	Correlational study	361	Beginning Engineering Students.	$r = 0{\cdot}48$ with performance in programming course.
	LeBold *et al.* (1983)	Correlational study	586		
Performance in college chemistry course	Barfield *et al.* (1983)	Correlation study	361 586	Beginning Engineering Students Beginning Engineering Students	$r = 0{\cdot}48$ with performance in programming course. $r = 0{\cdot}51$ with performance in programming course.
	Koubek *et al.* (1985)	Correlation study			
Performance in college physics course	Koubek *et al.* (1985)	Correlation study	586	Beginning Engineering Students	$r = 0{\cdot}48$ with performance in programming course.
Ability to tolerate stressful situations, adaptability to rapid changes, neatness	Weinberg (1971)	Observation	*	*	Suggests programming requires specific personality traits.

* No or insufficient data available.
† All correlations significant $P < 0.05$ level.

with performance in computer programming courses (Capstick, Gordon & Salvadori, 1975; Barfield, LeBold, Salvendy & Shadja, 1983; LeBold, Shadja, Salvendy & Lange, 1983). In one study (Koubek, LeBold & Salvendy, 1985), 25 to 50% of the variance in computer programming courses (FORTRAN and Pascal) was accounted for by mathematics, chemistry, and physics course performance. This seems to indicate that the cognitive skills required to learn programming share something in common with those used in learning mathematics and science concepts. Caution must be used, however, when attempting to extrapolate these results to expert programmers. The skills used to learn programming and those used when actually performing the programming task are probably not identical. Further research is needed to explore these differences.

Personality traits have also been suggested to identify differences in programmers. Weinberg (1971) identifies adaptability, neatness, and stress resistance as traits needed in programming. If this is true, these facts can be used as bits of information which characterize the cognitive tasks in programming and form a basis for a task analysis.

4. The learning process in programming

Another area that has yielded results useful in identifying and describing the cognitive tasks of programmers is the learning process. As seen earlier, experts first form a mental representation of the solution program while novices concentrate on specific functions of the problem. To aid novices, Mayer (1976) and Hoc (1977) have provided novices with a framework or overall structure of the program within which to begin the task. Novices provided with a framework performed better than the control group who were not provided with an overall structure. Another study by DuBoulay, O'Shea & Monk (1981), has suggested novices be shown how the computer processes the program through a "transparent notational machine". This is used to aid students in understanding the semantic concepts of a language. These studies, along with previously cited research, suggest that one of the first cognitive tasks in programming is to develop a structural framework, or a working model of the solution.

Other studies in learning programming have provided more direct clues to the programming tasks (Van der Veer & van de Wolde, 1983). Mayer (1979) has listed the levels of knowledge required in learning the BASIC programming language and the skills needed to use this knowledge. These skills include the ability to place each line of code, or statement, into its class or functional category, the ability to identify the events associated with each class of statements and the ability to chunk these classes into clusters. These skills identified by Mayer (1979) appear critical to the programming task, or at least to learning programming and are suggested as prerequisites for the programming task. Knowledge of these skills can be used to verify a task analysis methodology in that it should identify tasks which require these skills.

5. The relationship of programming subtasks to the cognitive aspects in programming performance

Examination of the learning process has lead to the identification of several skills required for execution of the cognitive tasks in programming. Similarly, examining the available literature on the subtasks of programming related to the cognitive processes involved can yield similar results. It is true that a programmer rarely, if ever, moves from the initial design phase to the final implementation of a program in a smooth sequence without repeatedly going back to various parts of the process for correction and modification. It is therefore probable that the subtasks within programming are interrelated and often dependent on each other. This is illustrated by the fact that it is difficult, if not impossible, to learn debugging without first knowing how to code a program. While the cognitive tasks performed in each programming subtask may be related, they are not necessarily identical and can be examined separately. For present purposes, dividing programming into subtasks, based on Shneiderman's (1980) framework, is convenient and advantageous for identifying specific cognitive components of each subtask.

5.1. REQUIREMENTS AND DESIGN

The first two subtasks a programmer faces in developing a program is to interpret the problem requirements and to design a procedure best suited to meet the requirements within given constraints. As Weinberg & Schusmon (1974) have shown, a tremendous amount of variance exists at this point due to varying interpretations of requirements. Programmers tend to interpret or emphasize components of the specifications differently. Also lending to this variance are differences in programmer experience in the application area and programmer creativity. Unless this variance is somehow controlled or otherwise accounted for, identifying common tasks will be extremely difficult.

Molhotra, Thomas & Miller (1980) identified three stages in designing a solution for any problem and suggested that these stages are also found in programming. These stages include goal elaboration, design generation, and design evaluation. During goal elaboration, subjects explicitly identify the goals and subgoals of the design. In design generation the requirements of the design, derived from the goals, are met by an interaction between the organization and elements of the design. The third stage, design evaluation, operates concurrently with the design generation phase and examines how well the design meets the various goals. Program design is extremely individualistic and therefore little research (with the exception of Adelson & Soloway, 1984; Jeffries, Turner, Polson & Atwood, 1981) has been done which can be used to identify the cognitive process involved.

Examination of design aids, particularly at a fine grain, have not helped identify the cognitive processes. For example, Shneiderman, Mayer, McKay & Heller (1977) found flowcharts to lead to little improvement over not using them. If useful design aids can be found, they will help in providing a clue as to what skills are necessary and what tasks require these skills. Currently, this subtask is ripe for innovative research strategies to extract the cognitive tasks involved.

5.2. CODING

The subtask most identified with programming and which follows the design phase is coding. Here the programmer converts the design into a programming language representation. In describing the cognitive processes of computer programming, some of the previously discussed models have attempted to characterize the cognitive processes in coding. Both Brooks' (1977) and Shneiderman's (1980) models are used to describe the coding process. With the groundwork already laid, enumerating the cognitive tasks in this subarea is most promising.

5.3. COMPREHENSION

Another area in which a theory has been developed, again by Brooks (1983) and Shneiderman (1980), is comprehension. This area also has been of particular interest to those researching chunking. Up to now, most chunking and comprehension experiments have used recall (Adelson, 1984; Kahney, 1983) of a program to gather information. However, comprehension of a program can exist at different levels and the skills used to comprehend or recall specific chunks may not be the same skills used to understand the overall function of a program. Again, different skills suggest different tasks. If this is true, identifying the tasks of comprehending a program have just begun and may require new methodologies to accompany recall. Also, further examination of the elements which make programs easier to comprehend, such as indentation (Miara, Musselman & Shneiderman, 1983) and control flow structures (Smith & Dunsmore, 1982) may help illuminate the cognitive tasks involved.

5.4. PROGRAM TESTING

While design, coding and comprehension are all supported by a particular model, the cognitive processes involved in program testing have been the focus of minimal research concerning the cognitive processes involved. Although this area may not be as rich or complex as other subtasks, it should be considered when analysing the overall programming task. Simply put, program testing checks to be sure the program does what it is supposed to do. This may be done by the author of the program, by teams of individuals (Myers, 1978) or by outside individuals. This subtask seems particularly well suited for such methodologies as the GOMS model since the alternatives are more limited. Also this subtask, more than the others, may lend itself to a standard procedure which would be effective for most programs.

5.5. DEBUGGING

On the opposite extreme, debugging procedures, which have been identified as the most difficult programming subtask (LeBold, Jagacinski, Koubek & Salvendy, 1985), do contain individual variation and have generated a great deal of research. While many studies (Nagy & Pennebaker, 1974; Youngs, 1974; Boies & Gould, 1974) have identified the types of errors, others (Gould, 1975) have attempted to develop a framework of the cognitive process of debugging and have identified the major tasks in debugging.

5.6. DOCUMENTATION

Another crucial subtask of programming is documentation. There is no question that this area is of great importance. However, there is much disagreement concerning good documentation requirements (Shneiderman, 1980). An analysis of this subtask has been more of an analysis of proper documentation techniques with less emphasis on the cognitive processes in documenting (Basili & Mills, 1982). This is probably the appropriate emphasis since proper techniques should be taught based on the results of a task analysis of the other subtasks, particularly comprehension.

5.7. MODIFICATION

The final subtask to be discussed, modification, is probably one of the most common programming tasks. Shneiderman (1980) indicates that some estimates suggest up to 75% of all programming work involves modifying old software-either to correct an error or to add new functions to a program. Modification requires a programmer to use the combined skills of comprehension, composition, and debugging, often in an unfamiliar environment. Because of its importance, and since little research concerning cognitive processes exists in this area (Sheppard, Borst, Curtis & Love, 1978), modification would be a next logical candidate for development of a model (Table 3 summarizes the current findings on programming subtasks).

6. Review of current cognitive task analysis methodologies

6.1. PROTOCOL ANALYSIS

A brief review of the methodologies used to develop the programming models and examine the subtasks of programming reveals quantitative, tightly controlled experimental studies and qualitative analysis. For qualitative analysis, protocol analysis has been a prominent methodology in problem solving research (Newell & Simon, 1972). In protocol analysis, subjects performing the task verbalize their thoughts as they solve the problem. This method, which includes interviews, is commonly used by knowledge engineers for extracting knowledge to build expert systems. Also, since protocol analysis has already been used as a methodology for deriving the cognitive processes of computer programmers, it is a good candidate for use in further examination of programming tasks.

Two major literature reviews representing different viewpoints concerning the usefulness of verbal reports have been compiled. The first study, done by Nisbett & Wilson (1977), reviews evidence that there may be little or no relation between verbal reports and higher order cognitive processes. In summarizing the existing literature and their own findings regarding human ability to verbally report cognitive processes, Nisbett & Wilson have produced four main conclusions. First, people are often unable to identify the existence of evaluative or motivational responses. Second, as observed when interviewing creative artists about their creative cognitive processes, people have difficulty even reporting that a process has occurred. Closely associated to this, Nisbett & Wilson's third point indicates that people often have difficulty identifying or acknowledging existence of critical stimuli. Their final point suggests that even if the stimulus and response are known, people cannot accurately explain the relationships between them. When people do give verbal reports, they are applying causal theories regarding the relationship of stimulus and response. In other words they simply make judgements "...about how plausible it is that the stimulus would have influenced the response" (Nisbett & Wilson, 1977).

Nisbett & Wilson (1977) are not unchallenged in their view of verbal reports. A major study by Ericsson & Simon (1980) has advocated the validity of these reports

TABLE 3
Research relevant to cognitive tasks in the programming subtasks

Subtask	Method of study	Reference	Results	Key unanswered questions
Reqts. and Design	Manipulation of independent variable	Weinberg (1974)	Large variance in design performance exists due to interpretation of problem statement.	Why do errors and individual differences arise and how are they accounted for in the model?
		Adelson & Soloway (1984)	Expert programmer design process identified.	
		Shneiderman (1977)	Flowcharts are of little value in program design	
Coding	Protocol Analysis	Brooks (1977)	Coding requires a large number of production rules.	Which cognitive skills are required in the coding process?
			Large variations in coding time (27:1) between similar tasks.	Why do large performance differences exist?
Comprehension	Recall, Protocol Analysis	Brooks (1983)	Comprehension consists of hypothesis generation and testing.	Which cognitive tasks are performed at different levels of comprehension?
		Adelson (1984)	Experts recall abstract knowledge first.	
		Kahney (1983)	Non-experts recall concrete knowledge first.	
Program testing and debugging	On-line protocol analysis	Bois & Gould (1974) Youngs (1974) Nagy & Pennebaker (1974)	Identification and classification of programming errors.	What skills are required to perform the cognitive tasks of debugging and how can these skills be measured?
	Manipulation of errors in stimuli program	Gould & Drongowski (1974) Gould (1975)	Evaluation of difficulty in identifying programming errors of varying types. Development of a cognitive model of debugging.	
	Recall and recognition	Weiser (1982)	Programmers divide large programs into coherent slices to aid debugging.	
Documentation	*Protocol analysis*	Basili & *Mills (1982)*	*Documenting a program is done in* incremental fashion from bottom-up.	*What are the cognitive tasks in* documentation?
Modification	Estimate	Shneiderman (1980)	Up to 75% of the programmer's job involves program modification.	What cognitive tasks are performed in modification and how do they relate to the cognitive tasks of the other programming sub-tasks?

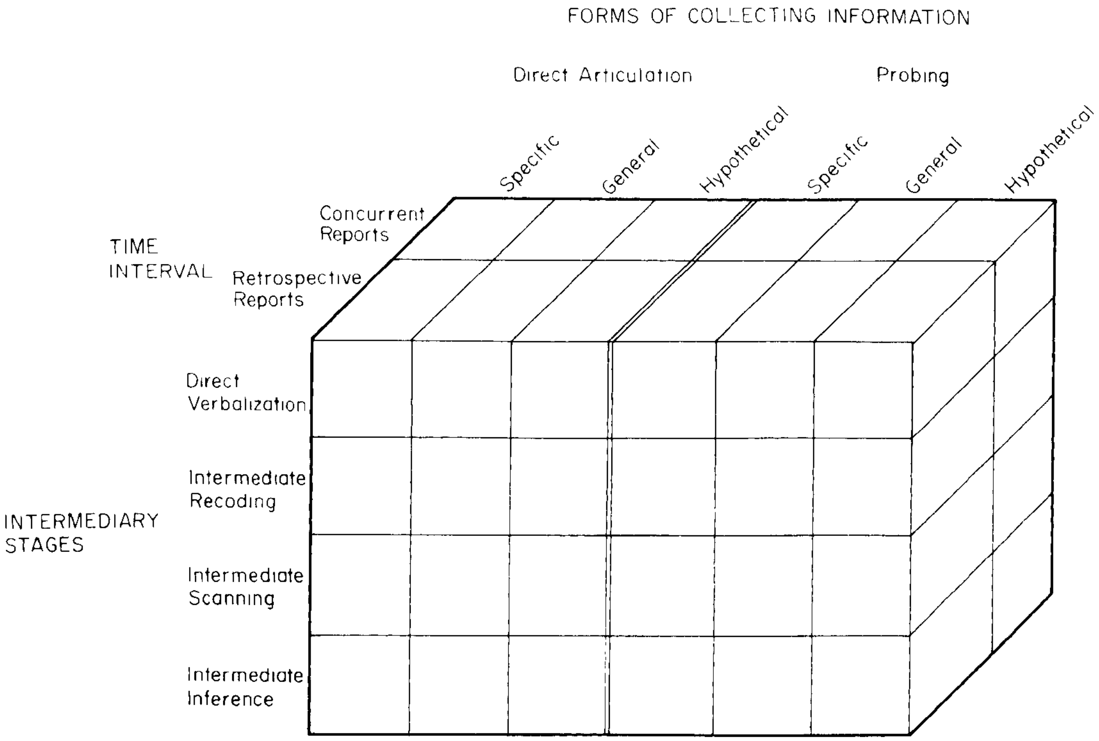

FIG. 2. Classification matrix for verbalization procedures

and attempted to resolve apparent conflicts with Nisbett and Wilson. They first point out that no clear guidelines exist regarding legitimate protocols. Here they make a significant contribution in developing protocol methodology by identifying four major factors useful in delineating the types of verbalization and the levels of recoding before verbalization (Fig. 2).

The first factor, Time Interval, has two levels; verbalization done concurrently with the task and verbalization performed retrospectively, after the task has been completed. This is similar to the time delay dimension discussed by Nisbett & Wilson (1977). The second factor, Form of Information Collection, incorporates situations where subjects can either "talk aloud" or respond to probing questions. The third factor delineates four intermediate stages of cognitive information recoding occurring between performance and verbalization. Level 1 corresponds to a direct transfer, when the information is reproduced in the same form in which it was acquired. Level 2 encoding occurs when the internal representation must be recoded into verbal form before reporting. Level 3 consists of an intermediate scanning substage, where the instructions require verbalization of only a selected part of the information the subject is using. Level 4, the inference process, occurs when the subject is required to verbalize something not normally noticed or used.

A fourth factor mentioned by Ericsson & Simon (1980) regards the generality of the reports. On one level, subjects can verbalize the details of the immediate trial being performed. On another, subjects can respond to hypothetical situations, and finally, they can verbalize in context free processes in which no situational details are given. For example, referring to Fig. 2, direct articulation of specific information in a concurrent time frame which requires intermediate recoding would take place if the subject described the tools he is using while performing the task. A retrospective report would require this description after the task had been performed. A probing report would require the subject to answer specific questions regarding the tools. In level 3, intermediate scanning would occur if the subject is asked to only describe features of the tool useful to the current task and intermediate inference may take place if the subject is asked how he controls the tool even though he had never considered it.

Ericsson & Simon (1980) propose consideration of the capabilities of the memory systems, Short Term Memory (STM) and Long Term Memory (LTM), and attention when evaluating verbal reports. Consistent with their predictions based on these capabilities, only when subjects are required to recode information does it become distorted and alter performance. Also, only information in "focal attention" can be verbalized. This includes only information to which the subject is currently attending and, therefore, verbalizing unattended information will produce guesses regarding the process and inductions rather than the actual process. As mentioned earlier, information processing becomes automatic and less available for verbalization as experience increases, resulting in inductions for the experimenter and/or subject to describe these automatic processes. Overall, however, Ericsson & Simon (1980) suggest that while some intermediate information may be missing, the obtained information is not completely invalidated (suggesting some information is better than no information).

When evaluating the use of protocols for programming task analysis, one immediately is confronted with how well the programming task fits into the category of tasks which will result in distorted or incomplete protocols. Also, to analyze enough protocols for sufficient reliability to be obtained is costly in both time and money. The best strategy for developing the task analysis methodology may be to first build on the strengths of existing methodologies. Thanks to the work of Ericsson & Simon (1980) and Nisbett & Wilson (1977) the potential areas of programming which can be evaluated by protocol may now be identified. After these areas are identified, other methods must be found to deal with the remaining areas of programming.

6.2. QUANTITATIVE METHODS IN COMPUTER PROGRAMMING

The methodology not affected by the weaknesses of subjective methods (such as protocol) is the quantitative, controlled experimental study. This method has been used often in exploring the cognitive components of programming subtasks and, therefore, should be considered as a tool for uncovering cognitive tasks in programming. In particular, the exploratory findings using other methods will most likely be subjected to confirmation in controlled experimental settings. Rather than reproduce a detailed analysis here, the reader is referred to the excellent reviews by Brooks (1980) and Shneiderman (1976).

6.3. EXPERTISE TRANSFER SYSTEM (ETS)

Aside from qualitative analysis of verbal reports and controlled experimental methods, it may be helpful to examine knowledge extraction techniques from other disciplines. John Boose (1986) has developed an interactive, computer-based methodology from the psychological theory of Personal Constructs by George Kelly (1955). This methodology, the Expertise Transfer System (ETS), elicits ratings on important components of the domain being analysed and combines them to form production rules (see Boose (1986) for details).

When evaluating ETS, Boose found it less effective in acquiring knowledge for synthesis type problems and better suited for analysis type problems, such as diagnosis and classification, which can be represented by production rules. He also points out a problem common to nearly all knowledge extraction methodologies; lack of certainty regarding completeness of information. The researcher can never be assured he has uncovered all relevant concepts for the problem space. However, since ETS was originally designed as a front-end processor to reduce extraction time, it still requires a knowledge engineer who can attempt to verify the completeness of the information.

The Expertise Transfer System appears to be a significant step in knowledge extraction by working from the broad spectrum of the initial interviewing to the final writing of production rules. Previously, systems needed the initial parameters and problem traits input by a human operator before beginning-information now provided by ETS. When applied to programming tasks, ETS can identify important components in the task but cannot tell how the task is performed. Therefore, if the task analysis only requires important traits in the task, ETS may be a viable alternative to protocol analysis, but if a model of programmer behavior is required, ETS can, at best, provide initial components and supplement other methods. At present, literature available on ETS is limited.

The previous discussion indicates a number of methodological issues which should be considered in the development of a task analysis for computer programming. Further, each potential methodology is best suited to a particular purpose and it is the researcher's responsibility to use his available resources and options in the most effective manner. If proper methodology is implemented, the results can be used to develop or support a theory of the cognitive processes of computer programming which could identify cognitive tasks and predict performance.

7. Recommendations and conclusions

Several task analyses of programming have already been performed and the methodologies used to develop and validate models of programmer behaviour have identified several cognitive subtasks in programming. Also, identification of the skills required by good programmers, through correlational studies and examination of the learning process, have suggested cognitive components which use these skills. Finally, examination of programming subtasks has revealed findings which, while not directly identifying cognitive tasks, does set boundaries in which these tasks can be found. However, to obtain the proposed benefits, a task analysis for programming must provide more specific and detailed information than that available from current research.

A review of extraction methodologies indicates that while controlled experiments and ETS avoid some of the biases and inadequacies of a subjective protocol, they do not collect relevant information with the same power as protocol analysis. Protocol analysis has already been used successfully for uncovering the higher level cognitive tasks of programming. However, this methodology has several weaknesses, including subjective interpretation, inaccuracy and limitations on subject types suitable for analysis (i.e., people who do not verbalize well are not useful for protocol analysis).

Several features can be incorporated into a task analysis procedure to overcome the weaknesses of current methodologies. For example, an effective analysis technique must not only elicit the process used in problem solving but do so in such a manner so as to avoid the distortions and incompleteness which currently plague traditional protocol analysis. The technique must collect information as the task is being performed to avoid any alterations due to time delay. This collection method should also gather information regarding the intermediate processes not normally stored in STM, and gain access to information not normally attended to without requiring subjects to make assumptions or theorize about the unattended information. All of this must be gathered in an unobtrusive manner so task performance is not altered.

Another feature required of this data gathering technique is the collection of complete information. When the cognitive load on the subject becomes too heavy and verbalization is either impossible or intrusive, another procedure must be available to collect information. Also, to identify relevant stimuli used by the subject, the method must monitor what information the subject is using when performing the task. This includes not only knowing what information is available from the external environment and prior learning, but what information is actually being perceived and used when performing the task.

A final requirement of such a system is the ability to provide an environment in which the effects of context and presence or absence of events can be examined. By examining the effect of events and non-events on task solution strategies in an experimental setting, the question regarding impact of non-events can be approached in a quantitative manner.

To fulfill these requirements, a proposed methodology, Computer Aided Protocol (CAP), has been developed (see Koubek, Salvendy, Eberts & Dunsmore, 1987 for details). It is suggested that CAP capitalize on the strengths of protocol analysis and ETS while compensating for their inherent shortcomings. CAP is implemented in an on-line interactive environment to obtain both the process used and declarative data. By recording all keystrokes and information viewed in a time sequenced manner, the CAP system collects data in an unobtrusive manner and overcomes the problem of non-verbalization during periods of heavy cognitive load. This information can be used to explore relationships between stimuli and responses, particularly for unattended events. While videotaping could also gather this information, the automatic recording procedure used here provides data in a format immediately suitable for analysis and reduces the subjectivity of analysis by eliminating one step of the interpretation process.

CAP is also designed to obtain goals and tasks on a higher level than keystrokes by presenting probing questions regarding these goals at logical points in the solution process. In its present implementation, CAP probes subjects after each input, requesting their reason for entering the command.

This methodology was tested with protocol analysis and a control group for their ability to extract information on the task of program modification. Thirty subjects (10 per group) were asked to modify an 87 line program and a randomly assigned extraction methodology (CAP, protocol analysis and a control group) was used. Results indicate that when compared to traditional protocol analysis, CAP obtained lower level goals and operators more accurately while protocol analysis was significantly more effective at collecting higher level goals and tasks ($F(1,8) = 11{\cdot}23$; $P < 0{\cdot}004$) (Koubek *et al.*, 1987). This suggests an integration of CAP and protocol for task analysis and knowledge in cognitive-oriented domains.

In conclusion, based on the rudimentary cognitive models presently available and the state of current methodologies for identifying cognitive tasks, an actual expert system based on human cognition to aid software development does not appear likely in the near future. However, significant research is underway (Boose & Gaines, 1986) to develop knowledge elicitation procedures for identification of cognitive tasks which can lead to cognitive models for an expert system. Another avenue of research which may help overcome this present limitation in expert system development is construction of systems based on an engineering model of the process, rather than trying to model the human process. While this approach will greatly reduce the demands made on knowledge elicitation tools, some type of elicitation procedure will still be required, further emphasizing the need for additional research.

This work was supported by the National Science Foundation under Grant No. MEA 8219688.

References

ADELSON, B. (1984). When novices surpass experts: the difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **10**(4), 483–495.

ADELSON, B. & SOLOWAY, E. (1984). *A Model of Software Design*. Yale Artificial Intelligence Laboratory, Yale University, New Haven, CT.

ANDERSON, J. R., GREENO, J. G., KLINE, P. J. & NEVES, D. M. (1981). Acquisition of problem solving skills. In J. ANDERSON, Ed. *Cognitive Skills and Their Acquisition*, pp. 191–230. Hillsdale, NJ: Lawrence Erlbaum Associates.

BARFIELD, W., KOUBEK, R. & HWANG, S. L. (1985). Eliciting expert behavior for software: A cognitive psychology approach. In R. EBERTS & C. EBERTS, Eds. *Trends in Human Factors/Ergonomics II*, pp. 303–311. Amsterdam: Elsevier.

BARFIELD, W., LEBOLD, W. K., SALVENDY, G. & SHADIA, S. (1983). Cognitive factors related to computer programming and software productivity. *Proceedings of the Human Factors Society, 27th Annual Meeting*, pp. 647–651.

BASILI, V. R. & MILLS, H. D. (1982). Understanding and documenting programs. *IEEE Transactions on Software Engineering*, **SE-8**(3), 270–283.

BOIES, S. J. & GOULD, J. D. (1974). Syntactic errors in computer programming. *Human Factors*, **16**(3), 253–257.

BOOSE, J. (1986). *Expertise Transfer for Expert System Design*. Amsterdam: Elsevier Science Publishing.

BOOSE, J. H. & GAINES, B. R. (1986). *Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

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

BROOKS, R. (1980). Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM*, **23**(4), 207–213.

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

Capstick, C. K., Gordon, J. D. & Salvadori, A. (1975). Predicting performance by university students in introductory computer courses. *SIGCSE Bulletin*, **7(3)**, 21–29.

Card, S. K., Moran, T. P. & Newell, A. (1980). Computer text-editing: An information-processing analysis of a routine cognitive skill. *Cognitive psychology*, **12**, 32–74.

Card, S. K., Moran, T. P. & Newell, A. (1983). *The Psychology of Human–Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chase, W. G. & Simon, H. A. (1973). Perceptions in Chess. *Cognitive Psychology*, **4**, 55–81.

DuBoulay, B., O'Shea, T. & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man–Machine Studies*, **14**, 237–249.

Ericsson, K. A. & Simon, H. A. (1980). Verbal reports as data. *Psychological Review*, **87(3)**, 216–254.

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

Gould, J. D. & Drongowski, P. (1974). An exploratory study of computer programming debugging. *Human factors*, **16(3)**, 258–277.

Hoc, J. M. (1977). Role of mental representation in learning a programming language. *International Journal of Man–Machine Studies*, **9**, 87–105.

Jeffries, R., Turner, A. A., Polson, P. G. & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson, Ed. *Cognitive Skills and Their Acquisition*. (pp. 255–284). Hillsdale, NJ: Erlbaum.

Kahney, H. (1983). Problem solving by novice programmers. In T. R. G. Green, S. J. Payne & G. C. van der Veer, Eds. *The Psychology of Computer Use*, pp. 121–141. London: Academic Press.

Kelly, G. (1955). *The Psychology of Personal Constructs*. New York: Norton.

Kessel, K. L. (1984). Task analysis in applying software design principles. In E. Grandjean, Ed. *Ergonomics and Health in Modern Offices*. pp. 170–174. London: Taylor & Francis.

Koubek, R. J., LeBold, W. K. & Salvendy, G. (1985). Predicting performance in computer programming courses. *Behavior and Information Technology*, 113–129.

Koubek, R. J., Salvendy, G., Eberts, R. E. & Dunsmore, H. (1987). Eliciting knowledge for software development. *Behaviour and Information Technology*, **6(4)**, 427–440.

LeBold, W. K., Jagacinski, C., Koubek, R. & Salvendy, G. (1985). Assessing computer programming at the university level. *Advanced Systems for Manufacturing 12th Conference on Production Research and Technology*, pp. 459–468. Washington, D.C.: National Science Foundation.

LeBold, W., Shadja, S., Salvendy, G. & Lange, T. (1983). Cognitive and affective factors related to computer programming performance. *IEEE Frontiers in Education Conference Proceedings*, 53–61.

Mayer, R. (1976). Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. *Journal of Educational Psychology*, **68(2)**, 143–150.

Mayer, R. (1979). A psychology of learning BASIC. *Communication of the ACM*, **22(11)**, 589–593.

McCormick, E. J., Jeanneret, P. R. & Mecham, R. C. (1969). Position Analysis Questionnaire. Purdue Research Foundation, West Lafayette, Indiana, 47907.

McKeithen, K. B., Reitman, J. C., Rueter, H. H. & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**, 307–325.

Miara, R., Musselman, J. & Shneiderman, B. (1983). Program identification and comprehensibility. *International Journal of Man–Machine Studies*, **12**, 119–140.

Molhotra, A., Thomas, J. C. & Miller, L. A. (1980). Cognitive processes in design. *International Journal of Man–Machine Studies*, **12**, 119–140.

Myers, G. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, **21(9)**, 760–768.

NAGY, G. & PENNEBAKER, M. C. (1974). A step toward automatic analysis of student programming errors in a batch environment. *International Journal of Man–Machine Studies*, **6**, 563–578.

NEWELL, A., & SIMON, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

NISBETT, R. E., and WILSON, T. D. (1977). Telling more than we know: Verbal reports on mental processes. *Psychological Review*, **84**, 231–259.

SHEPPARD, S. B., BORST, M. A., CURTIS, B. & LOVE, T. (1978). Predicting programmer's ability to understand and modify software. *Symposium Proceedings: Human Factors and Computer Science*, pp. 115–135. Washington, D.C.

SHNEIDERMAN, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Systems*, **5(2)**, 123–143.

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

SHNEIDERMAN, B., MAYER, R., MCKAY, D. & HELLER, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, **20(6)**, 373–381.

SMITH, C. & DUNSMORE, H. (1982). On the relative comprehensibility of various control structures by novice Fortran programmers. *International Journal of Man–Machine Studies*, **17**, 115–117.

VAN DER VEER, G. C. & VAN DE WOLDE, G. J. E. (1983). Individual differences and aspects of control flow notations. In T. R. G. GREEN, J. J. PAYNE & G. C. VAN DER VEER, eds. *The Psychology of Computer Use*. pp. 107–120. London: Academic Press.

WEINBERG, G. M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

WEINBURG, G. M. & SCHUSMON, E. L. (1974). Goals and performance in computer programming. *Human Factors*, **16(1)**, 70–77.

WEISER, & SHNEIDERMAN, B., (1982). Human factors of computer programming. In G. SALVENDY, Ed. *Handbook of Human Factors*. pp. 1392–1416. New York: John-Wiley.

YOUNGS, E. A. (1974). Human errors in programming. *International Journal of Man–Machine Studies*, **6**, 361–376.