A REVIEW OF HUMAN FACTORS RESEARCH ON PROGRAMMING LANGUAGES AND SPECIFICATIONS

Bill Curtis
ITT Programming Technology Center
1000 Oronoque Lane
Stratford, CT 06497

ABSTRACT

This paper presents a partial review of the human factors work on computer programming. It begins by giving an overview of the behavioral science approach to studying programming. Because of space limitations this review will concentrate on cognitive models of programmer problem solving and the experimental research on language characteristics and specification formats. Areas not reviewed include debugging, programming teams, individual differnces, and research methods. The conclusions discuss promising directions for future theory and research.

I. INTRODUCTION

Although most seminars at local motels will claim to improve programmer performance, the scientific demonstration of these improvements is the province of behavioral scientists. Experimental research is the method of choice among behavioral scientists for empirically demonstrating their claims (Curtis, 1981). The purpose of this review is to present a broad overview of the psychological research on programming.

The following areas have been of primary relevance in developing a psychological understanding of programming:

- Cognitive ergonomics
- Cognitive psychology
- Psycholinguistics
- Industrial/organizational psychology

Table 1 lists some of the areas which could be studied through the paradigms of each of these four areas of psychology. These contributions are listed by phase of the software life cycle.

Many of the topics listed in Table 1 have not been addressed. In the absence of empirical evidence, computer scientists must use experience and judgment in designing and using requirements, design, or programming languages. While experience helps refine existing languages, it does not always suggest new alternatives. Further, such languages are often designed from a primarily engineering orientation with less balance being supplied from a human factors analysis.

TABLE 1

Potential Psychological Research on Programming

Area of Psychology	Requirements	Design	Coding	Verification	Maintenance
Cognitive Ergonomics	Requirements tools and methods	Design tools and methods	Coding tools	Debugging & test tools	Design, coding debugging & test tools
Cognitive	Identifying the full problem space	Linking problems to technical solutions	Translation processes	Dissection strategies	Program comprehension
Psycholin-guistics	Requirements languages	Design languages	Computer languages	Cues for error detection	System documentation languages
Industrial/ organizational	analyst/ customer interaction	Personnel selection & training	Team performance	Interteam dynamics	Motivation and career planning

II. MODELS OF PROBLEM SOLVING IN PROGRAMMING

In order to realize psychology's implications for software development, it is important to understand some psychological models of the cognitive processes underlying programming tasks. The models presented here are based on cognitive theories which are still undergoing development and elaboration in psychological research.

Most models of programmer problem solving begin by distinguishing between short and long term memory. Short term memory is a limited capacity workspace that holds and processes those items under attention. Long term memory is a store which retains knowledge over long time periods without our being consciousness of it.

Since G. A. Miller published his 1956 article on "the magic number seven, plus or minus two", short term memory has been characterized as being able to retain about seven items at a time. While this is a fair characterization of a simple mental phenomenon, the magic number seven has found its way into explanations it was never intended to support (cf. Halstead, 1977). The magic number seven should not be used to argue that a sub-program should consist of no more than seven modules, or that a team leader's effective span of control is about seven members. These phenomena are affected by processes unrelated to those under which the magic number seven was validated.

Nevertheless, this limit to the capacity of short term memory is one of the most significant factors limiting our ability to develop large systems quickly and accurately. Modularization and information hiding are two techniques for limiting the amount of information a programmer must keep track of at one time. By breaking a problem into smaller portions a programmer is able to reduce it to a size that can be attended to at one time.

A process called chunking expands the capability of our short term mental workspace. In chunking, several items are bound together conceptually to form a unique item. In programming the concept of a chunk is extremely important. Through experience and training programmers are able to build increasingly larger chunks based on frequent patterns which emerge in solving problems. Much of a programmer's maturation involves observing more patterns and building larger chunks. The nature of the concepts that a programmer has been able to build into a chunk provides an indication of his/er ability. The nature of the elements chunked together has important implications for educating programmers. Educational material and exercises should be presented in a way which maximize the likelihood of building useful chunks.

Long term memory is usually treated as having limitless capacity for storing information. The important consideration with long term memory is the way information is stored. It is important that information be related and indexed so that appropriate chunks are recalled quickly and accurately. The structure of memory and the associations developed among memory chunks are important for any theory of how programming knowledge develops and is used.

Shneiderman and Mayer (1979) presented a model using short term, long term, and working memories. The major contribution of their model is the distinction in long term memory between semantic and syntactic knowledge. Semantic knowledge concerns general programming concepts or relationships in the applications domain which are independent of the programming language in which they will be executed. Syntactic knowledge involves the procedural particulars of a given language. Their model describes how long term memory for programming concepts is organized at a global level.

Brooks (1977) approached programming from the perspective of Newell and Simon's (1972) model of human problem solving and its later elaboration into production systems by Newell (1973). Production systems are a set of stimulus/response-like relations between conditions which can be satisfied by events in short term memory and actions which are events occurring in long term memory. The rules of programming built up through production systems can be modeled in a way similar to Petri nets. Thus, Brooks shows how to build a structure in long term memory with specific types of knowledge in it.

McKeithan, Reitman, Rueter, and Hirtle (1981) have demonstrated qualitative differences between expert and novice programmers. In their first experiment, McKeithan, et al. found that expert programmers could recall a greater portion of a memorized modular program than could novices. However, this effect disappeared when the lines of the program were randomly scrambled prior to their presentation. In a second experiment they demonstrated that some expert-novice differences could be accounted for by the schemas that programmers learn to use in organizing the knowledge they gain with experience. In particular, McKeithan et al. showed that experts held similar schema for language commands in Algol. Intermediate programmers had schema that resembled those of experts more than those of other intermediates. The organizations exhibited by novices were idiosyncratic and did not appear to be based on the logical structure of the language.

Mayer (1981) describes several training techniques grounded in some of the notions underlying expert-novice differences. The understanding of expert/novice differences is critical when interpreting research conducted on undergraduates versus career programmers. Undergraduates and other novices do not have the same structures (often called schemas) built up in long term memory as do experienced programmers.

Thus, their reasoning about programming is qualitatively different from that of more experienced programmers.

Atwood, Turner, Ramsey, and Hooper (1979) have used Kinstch's (1974) propositional structure of memory to demonstrate how higher order memory chunks (macro-propositions) are built by programmers. In brief, Kintsch argued that any text base (e.g., a program text) could be understood as a series of micropropositions. A microproposition is composed of a relational concept (operator) and one or more arguments (operands). The referential coherence of the program depended on the overlap of arguments between propositions. Where such overlap did not exist, it might be inferred. The macrostructure of the program is built as schemas direct the development of macropropositions from the many micropropositions.

Atwood, et al. demonstrated that the use of these schema allows more experienced programmers to recall a program in greater depth than a novice is capable of. Atwood and Ramsey (1978) further demonstrated in a debugging study that bugs in code which were closer to a macropropositional statement (a statement under which other statements could be hierarchically nested in an and/or tree diagram of the program) were easier to find than those in statements further down in the hierarchy. These macropropositions are similar to the notion of advanced organizers that Mayer (1981) recommended for use in training programmers.

In summary, the limitations of human memory provide severe constraints for programmers. The limits of short term memory are overcome by chunking information at increasingly higher levels of abstraction. Chunking includes the types of processes that have been described as developing macropropositions and using advanced organizers in learning.

As programmers structure their knowledge through the chunking of programming concepts, their understanding of programming changes qualitatively. Expert programmers show a greater consensus in the structure of their programming knowledge than do novices. The chunks that constitute this structure can be retrieved and integrated when solving problems. Thus, while the novice may be working at level of lexical elements, the expert is working at the level of chunks composed of numerous lexical elements (or of a hierarchy of chunks the lowest level of which are composed of lexical elements). Once embodied in a design or code, these chunks are represented as macropropositions which help organize a programmer's understanding of the program.

III. LANGUAGE CHARACTERISTICS

The Whorfian hypothesis suggests that people's ability to think is limited by the language they are thinking in. That is, the structures a language presents for manipulating words and the vocabulary available for representing ideas constrains the thoughts that can be easily and accurately represented. The enormous variation in symbols, constructs, and syntax observed among natural languages is also true of computer languages. Computer language differences range from the long compound words of Cobol to the symbolic brevity of APL, from the massive size of PL/1 to the compactness of Pascal. Language structure has always fascinated computer scientists, and it is not surprising that the largest body of research on human factors in software development has emerged from this area.

Interest in specific language structures began seriously when Dijkstra assailed the GO TO statement in 1968. He argued that having logic wander through programs as it often does with GO TOs made them difficult to develop correctly and even more difficult to understand. This argument is psycholinguistic in that it is predicated on the interaction between human problem solving/comprehension and language structure. The structured programming movement emerged calling for greater discipline in constructing control logic and specifying control structures to be included in future languages. Although the structured programming movement involved many more changes in software engineering practice than a discipline for coding, structured coding came to represent this evolution in programming practices.

A number of computer scientists began conducting experiments to demonstrate the value of structured coding. One of the earliest was a dissertation by Larry Weissman (1974) at the University of Toronto. Although his results were only suggestive, this was the first study to take a behavioral perspective on how program characteristics affect programmers. Subsequently, an experiment by Lucas and Kaplan (1974) demonstrated that structured programs were easier to modify.

The first program of research on control constructs appears to have been established by Max Sime, Thomas Green, and their associates at the University of Sheffield (Sime, Arblaster, & Green, 1977; Green, 1977; Green, Sime, & Fitter, 1980). Their research represents the entry of cognitive ergonomics into the study of programming. Through research publications that have spanned almost a decade they have demonstrated among other things:

- the superiority of nested over GO TO conditional statements,
- problems with certain ways of nesting conditionals,
- the advantages of redundant expression of controlling conditions at the entrance of each conditional branch,

- the benefits of a software task vary with the nature of the programming task, and
- a standard (perhaps automated) procedure for generating the syntax of a conditional statement can improve coding speed and accuracy.

This work was followed by a program of research initiated at GE by Tom Love and continued by Sylvia Sheppard, Phil Milliman, and myself on the benefits of structured code in a series of programming tasks (Sheppard, Curtis, Milliman, & Love, 1979). We found that it was the general discipline of a top-down flow of control that was important rather than the specific procedural details of how the code was structured.

John Gannon and Hubert Dunsmore conducted a program of research into numerous language features. In his first study, Gannon (1979) modified TOPPS (a procedure oriented language) to improve characteristics which were believed to cause difficulty in constructing programs. His results demonstrated that these language modifications resulted in fewer errors of the types made with the original constructs in TOPPS. However, there were no differences in overall error occurence. These results argue that changes in language characteristics can have an effect on programming, but this is not necessarily a general effect on overall performance. The effect is limited to the specific kinds of problems experienced with the original language feature.

In a second study, Gannon (1977) compared programming effectiveness in a statically typed and a typeless language. He found more errors being committed in the typeless language. He did not find evidence that the increased effectiveness of the statically typed language was due to the compile time checking of data types. Rather, this effectiveness stemmed from the increased power this language possessed from eliminating some of the programmer's concerns about the machine representation of data. The benefits of statically typed languages seemed to accrue mainly on less experienced and capable programmers.

In a subsequent analysis of these data, Dunsmore and Gannon (1979) studied the effects of variable referencing. They defined a variable as 'live' from its first to its last reference in a procedure. They observed a range of from 0.57 to 2.06 variables per statement. Programs exhibiting moderate levels of variable referencing (1.51 to 1.70 variables per statement) required less effort to produce. Dunsmore and Gannon also found that modifications were easier to make in programs with fewer live variables per statement.

An early program of research was instituted by Lance Miller and his colleagues at IBM's Thomas J. Watson Research Center. This program investigated a broad spectrum of topics, and several of the resulting papers dealt with language issues. In particular, Miller has been concerned with the use of natural language commands by people who are not familiar with a computer language. In his first study, Miller (1974) demonstrated that 'or' problems were more difficult to handle than 'and' problems, and that negation increased the difficulty of a problem to be solved.

In his second study, Miller (1981) studied how non-computer science students expressed procedural specifications in English for 6 separate file searching problems. He observed that they used a common personal approach in specifying all 6 problems, even though the problem structures differed. They had difficulty in being explicit about procedures and assumed that variable references were clear from the context of their paragraph. They typically used a small vocabulary with short descriptions, but the size of this vocabulary could be reduced even further when redundancies were removed. The students spent more time with data manipulation and less time with transfer of control. This is an important point. Programming languages provide massive control structures with embedded data manipulations. However, the natural human tendency seems to begin with data manipulations and add control structure as a qualification to the action. Miller concluded that natural language was not adequate for procedural specifications, but that a constrained subset might be more effective.

Two primary themes seem to have emerged in the research of language structures. First, structuring the control flow assists programmers in understanding a program. Structured programming allows programmers to develop systematic expectations about what is happening in a program. Thus, as they develop macropropositions for organizing their understanding, programmers are able to concentrate more of their cognitive resources on the semantic content of the algorithm, since fewer resources are required to track the direction of control flow. In addition, a structured program is more easily broken into sensible units as a programmer develops a hierarchical structure of macropropositions to represent the semantic content of the program.

The second theme emerging from this research is the importance of the data structure. Gannon found that the advantage of strongly typed languages appeared to reside in the enhanced data structures they supported. Miller indicated that people tend to think in terms of data manipulations rather than control structures. The areas of data structure and use have only been lightly considered in psychological research on programming. However, there are clear suggestions that this will prove a fertile area for further study. Concentration on data structure and flow may provide a better integration of programming tools and procedures with the structure of human cognition.

There are a number of issues which have not

been addressed in a systematic program of research on language design and use. Some of these are:

- the size and versatility of a language versus its learnability and ease of use,
- the tradeoffs to be made in data versus control structures,
- the use of procedural versus non-procedural languages,
- the effect of the first language learned on learning and using other types of languages, and
- the design of requirements and specification languages.

If they do not act quickly, psychologists will have missed an excellent opportunity to make significant contributions to the development of advanced programming techniques.

IV. SPECIFICATION FORMATS

The amount of interest in specification formats has not rivaled that in language characteristics. This is unfortunate since the most frequent and expensive faults made in programming large systems are created during the design phase. Characteristics of control structures are not important issues until the detail design phase at the earliest. The requirements and preliminary design of the system are far more crucial to the ultimate success of the development effort than are detail design and coding. Given the sensitivity of a programming project to the initial specifications, it is important to insure that the format of these documents is easy for programmers and analysts to comprehend.

Research in cognitive psychology has demonstrated that the way a problem is represented can have profound impact on someone's ability to solve it. This effect occurs because different ways of representing problems enhance different features of the problem which may be more or less relevant to solving it.

Research on specification formats in software engineering emerged in response to the debate over the use of flowcharts. Much of this debate had less to do with the actual format of flowcharts than with the difficulty of modifying them manually or he fact that they were often developed in the same unstructured manner as the code they documented. These problems deal more with the semantic content of what is documented in flowcharts than with the syntactic problem of how this information is organized and represented.

Four seminal studies outside of the programming area were performed in the mid-1970's comparing flowcharts to other formats for arriving at decisions. In the first of these, Wright and Reid (1973) compared prose, short sentences, decision tables, and tree charts in solving travel problems with time, cost, distance, and mode tradeoffs. For easy problems they found phrases, tables, and trees to be of equal value and superior to prose for making correct decisions. For more difficult problems, however, the trees were superior to all other formats.

In a similar study, Blaiwes (1974) found that adding a short verbal cue to decision points in a tree chart enhanced its effectiveness on difficult problems. He also found evidence replicating Wright and Reid's finding that when working from memory, information is better recalled when its presentation originally involved verbal encoding.

Kammann (1975) compared the effectiveness of telephone dialing instructions presented in either prose or two forms of flowcharts whose decision structures were arranged either broad or deep. He found both flowchart formats to be more effective than the prose. Broad flowcharts were especially effective for naive users.

Mayer (1976) represented some decision criteria in both verbal and flowchart form for determining winnings based on the outcome of a three team tournament. Taking a lead from Sime and Green, Mayer expressed the decision content in either IF-GOTO, shortened IF-GOTO, IF-THEN-ELSE, or decision tree structures. Mayer found interesting interactions between decision structure and representational format. These interactions led him to conclude that diagrams help to locate relevant information primarily where it was difficult to locate in verbal form. However, diagrams were not as effective as well structured verbal material. He also concluded that the representational format does not tend to influence the reasoning process separate from its effect on quickly identifying relevant information.

The initial, provocative study suggesting that flowcharts were of little use in a number of programming tasks was published by Shneiderman, Mayer, McKay, and Heller in 1977. In a series of five experiments on the composition, comprehension, debugging, and modification of programs, Shneiderman et al. found little benefit derived from flowcharts if code listings were available. However, there was a suggestion in the third experiment that the benefits of flowcharting are dependent on previous experience with flowcharts. Most of the students Shneiderman et al. studied were novices, and it is important to consider how these results might differ with a more experienced group.

Ramsey, Atwood, and Van Doren (1978) compared flowcharts to a program design language (PDL) and found no differences in code implementation. However, they found that designs

expressed in PDL were more detailed and complete than those expressed in flowcharts. In part these differences may reflect the cumbersomeness of developing flowcharts.

Brooke and Duncan (1980, 1981) performed three studies, the first of which supports Shneiderman et al's. findings that flowcharts provided little assistance in finding program faults. However, they believe that flowcharts did assist novices in following the program's decision structure, although they received little benefit in determining its correctness. The second and third experiments strengthen and extend these results. From these studies they concluded that flowcharts may help avoid irrelevant questions in performing tests, but that they did not help eliminate characteristic debugging mistakes. The advantages of flowcharts diminished as one moves from tracing flow to comprehending relationships between separate procedures.

Sylvia Sheppard, Elizabeth Kruesi, and I initiated a program of research into the characteristics of specification formats which affected their usefulness in different programming tasks. We distinguished between three types of symbology used in specifications: natural language, constrained language (PDL), and ideograms (flowchart symbols). We also distinguished between three types of spatial arrangements for the information presented: sequential, branching (flowchart-like), and hierarchical. These two dimensions are independent, but in comparing flowcharts to PDL or some other format it is difficult to determine whether any differences observed are due to symbology or to spatial arrangement.

The results of the first two studies (comprehension and coding) in a series of four (Sheppard, Kruesi, & Curtis, 1981; Sheppard & Kruesi, 1981) indicated that natural language is a poor way to present procedural information. Data in the comprehension experiment indicate some advantages to the branching, and in one instance the hierarchical arrangement. This spatial effect occurred on tasks where control flow information was important. On a task in which data flow information seemed more relevant, the effect did not occur. Similarly in the coding study, fewer control flow errors were made when working from a specification presented in a branching format. However, the development of a correctly running module was performed more quickly with fewer submissions when working from constrained language presented in either sequential or branching arrangement.

In a recent series of experiments, Shneiderman (1981) considered the alternative of data structure specifications. In his first study, Shneiderman found that data structure documentation added significantly to program comprehension, even when provided in conjunction with an input specification and pseudocode. In his second study Shneiderman compared control flow and data structure information presented in both textual and graphic formats. He found that although program comprehension did not differ between the textual and graphic formats, it was significantly better when working from the data structure rather than the control flow information.

In summary, two important themes have emerged from the research on specification formats. First, although flowchart-like formats have been shown to be effective in aiding a decision making task, they have not been shown to be as useful in programming as other forms of software specifications such as a program design language. The results from research on software specifications generally confirm Mayer's contention that diagrammatic notations are not as effective as well structured verbal material. A branching or hierarchically arranged format contributes significantly to understanding only in those situations where control flow information is critical (Fitter & Green, 1979). The frequency of these situations may not be as ubiquitous as the flood of control flow documentation would suggest.

Second, as suggested in the language section, Shneiderman's recent results suggest that data structure is more important than control flow in imparting information about a program. Thus, specification formats which highlight control flow will prove valuable much less often than formats which emphasize data structure and organization. Much more work needs to be directed toward the development and evaluation of specification formats which highlight the data structure and flow.

V. CONCLUSIONS

The structure of most programming tasks is inconsistent with how most people process procedural problems. Miller argued that computer languages are masses of control flow with embedded data manipulations, while people think about data manipulations and attach control flow. Results in the previous sections highlight three important principles. First, comprehension is aided by a structured control flow. Second, languages should provide the capacity to handle a broad range of data structures. Third, specifications should highlight the information relevant to a programming task, especially the data structure and flow.

Most procedural languages do not capitalize on the natural cognitive tendencies of programmers. Human factors research is critical in guiding and evaluating developments in programming technology. The principles above jointly insure that control flow does not enmesh programmers in procedural problems, and that they are free to develop a solution with elements (data structures) that are more suited to manipulation by their own style of problem solving. Methods for pursuing research in these areas are discussed in Brooks (1980), Curtis (1980), and Moher and Schneider (1981).

REFERENCES

Atwood, M.E. & Ramsey, H.R. Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging (Tech Rep. TR-78-A21). Alexandria, VA: ARI, 1978).

Atwood, M.E., Turner, A.A., Ramsey, H.R., & Hopper, J.N. An exploratory study of the cognitive structures underlying the comprehension of software design problems (Tech Rep. 392). Alexandria, VA: ARI, 1979.

Blaiwes, A.S. Formats for presenting procedural instructions. Journal of Applied Psychology, 1974, 59, 683-686.

Brooke, J.B. & Duncan, K.D. An experimental study of flowcharts as an aid to identification of procedural faults. Ergonomics, 1980, 23 (4), 387-399.

Brooke, J.B. & Duncan, K.D. Experimental studies of flowchart use at different stages of program debugging. Ergonomics, 1980, 23 (11), 1057-1091.

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

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

Curtis, B. Measurement and experimentation in software engineering. Proceedings of the IEEE, 1980, 68 (9), 1144-1157.

Curtis, B. Human Factors in Software Development. Silver Spring, MD: IEEE Computer Society, 1981.

Dijkstra, E.W. GO TO statement considered harmful. Communications of the ACM, 11 (3), 147-148.

Dunsmore, H.E. & Gannon, J.D. Data referencing: An empirical investigation. Computer, 1979, 12 (12), 50-59.

Fitter, M. & Green, T.R.G. When do diagrams make good computer languages? International Journal of Man-Machine Studies, 1979, 11, 235-261.

Gannon, J.D. An experiment for the evaluation of language features. International Journal of Man-Machine Studies, 1976, 8, 61-73.

Gannon, J.D. An experimental evaluation of data type conventions. Communications of the ACM, 1977, 20 (8), 584-595.

Green, T.R.G. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 1977, 50, 93-109.

Green, T.R.G., Sime, M.E., & Fitter, M.J. The problem the programmer faces. Ergonomics, 1980, 23 (9), 893-907.

Halstead, M.H. Elements of Software Science. New York: Elsevier, 1977.

Kammann, R. The comprehension of printed instructions and the flowchart alternative. Human Factors, 1975, 17 (2), 183-191.

Kinstch, W. The Representation of Meaning in Memory. Hillsdale, NJ: Erlbaum, 1974.

Lucas, H.C. & Kaplan, R.B. A structured programming experiment. The Computer Journal, 1974, 19 (2), 136-138.

Mayer, R.E. Comprehension as affected by the structure of the problem representation. Memory & Cognition, 1976, 4 (3), 249-255.

Mayer, R.E. The psychology of how novices learn computer programming. ACM Computing Surveys, 1981, 13 (1), 121-141.

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

Miller, G.A. The magic number seven, plus or minus two. Psychological Review, 1956, 63, 81-97.

Miller, L.A. Programming by non-programmers. International Journal of Man-Machine Studies, 1974, 6, 237-260.

Miller, L.A. Natural language programming: Styles, strategies, and contrasts. IBM Systems Journal, 1981, 20 (2), 184-215.

Moher, T. & Schneider, G.M. Methods for improving controlled experimentation in software engineering. Proceedings of the Fifth International Conference on Software Engineering. Silver Spring, MD: IEEE Computer Society, 1981, 224-233.

Newell, A. Models of production systems. In W.G. Chase (Ed.), Visual Information Processing. New York: Academic, 1973, 463-526.

Newell A. & Simon, H.A. Human Problem Solving. Englewood Cliffs, NJ: Prentice-Hall, 1972.

Nichols, J.A. Problem solving strategies and organization of information in computer programming. Dissertation Abstracts International, 1981.

Ramsey, H.R., Atwood, M.E., & Van Doren, J.R. Flowcharts vs. program design languages: An experimental comparison. Proceedings of the 22nd Annual Meeting of the Human Factors Society. Santa Monica, CA: Human Factors Society, 1978, 709-713.

Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Modern coding practices and programmer performance. Computer, 1979, 12 (12), 41-49.

Sheppard, S.B. & Kruesi, E. The effects of the symbology and spatial arrangement of software specifications in a coding task. Trends and Applications 1981: Advances in Software Technology. Silver Spring, MD: IEEE Computer Society, 1981.

Sheppard, S.B., Kruesi, E., & Curtis, B. The effects of symbology and spatial arrangement on the comprehension of software specifications. Proceedings of the Fifth International Conference on Software Engineering. Silver Spring, MD: IEEE Computer Society, 1981, 207-214.

Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop, 1980.

Shneiderman, B. Two experiments on control flow and data structure documentation (unpublished manuscript). College Park, MD: Department of Computer Science, University of Maryland, 1981.

Shneiderman, B. & Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 1979, 8 (3), 219-238.

Shneiderman, B., Mayer, R., McKay, D., & Heller, P. Experimental investigations of the utility of detailed flowcharts in programming. Communications of the ACM, 1977, 20 (6), 373-381.

Sime, M.E., Arblaster, A.T., & Green, T.R.G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-216.

Weissman, L. Psychological complexity of computer programs: An experimental methodology. Sigplan Notices, 1974, 9 (6), 25-36.

Wright, P. & Reid, F. Written information: Some alternatives to prose for expressing the outcome of complex contingencies. Journal of Applied Psychology, 1973, 57, 160-166.