FIFTEEN YEARS OF PSYCHOLOGY IN SOFTWARE ENGINEERING: INDIVIDUAL DIFFERENCES AND COGNITIVE SCIENCE

BILL CURTIS

Microelectronics and Computer Technology Corporation (MCC)
Austin, Texas

ABSTRACT

Since the 1950's, psychologists have studied the behavioral aspects of software engineering. However, the results of their research have never been organized into a subfield of either software engineering or psychology. This failure results from the difficulty of integrating theory and data from the mixture of paradigms borrowed from psychology. This paper will review some of the psychological research on software engineering performed since the Garmisch Conference in 1968. This review will be organized under two of the psychological paradigms used in exploring programming problems: individual differences and cognitive science. The major theoretical and practical contributions of each area to the theory and practice of software engineering will be discussed. The review will end with a call for more research guided by the paradigm of cognitive science, since such results are the easiest to integrate with new developments in artificial intelligence and computer science theory.

PARADIGMS IN SEARCH OF A FIELD

Since the 1950's psychologists have studied the behavioral aspects of software engineering. However, the results of their research have never been organized into a subfield of either software engineering or psychology. This failure results from the difficulty of integrating theory and data from the mixture of paradigms borrowed from psychology. The behavioral studies performed by computer scientists have been criticized by Brooks (1980) and Sheil (1981) for a lack of experimental rigor. Although they occasionally scoured the fine print to uncover something about each study to condemn, the gist of their remarks is well taken.

Every psychological study portrays a paradigm, a model of what the investigator believes is really important in human behavior. When the choice of paradigms is unconscious, investigators are often faced with defending their hypotheses with data which do not address the argument. The motley body of psychological studies on programming has been guided by numerous psychological paradigms, among them individual differences, human factors, cognitive science, group behavior, and organizational behavior. These paradigms represent different ways of looking at human beings, and differ in the aspects of human behavior they explain. Due to the limited space in the conference proceedings, only contributions from the individual differences and cognitive science areas will be reviewed. See Curtis (1981a,b) for reviews of research from other psychological paradigms.

Individual Differences

In the beginning was the need to hire the best person for the job. Programmers had always differed from each other in large ways, especially in their ability to write programs which optimized the precious resources of the machine. Since programming was a mental activity, it stood to reason that tests of cognitive ability should predict who would make the best programmers. However, measurement was difficult when the phenomena underlying the performance of a skill were unobservable, such as with mental abilities.

To measure mental abilities, a task must be devised which exercises the theoretical mental construct. The critical factors for this approach are:

1) a clear definition of the mental construct,
2) a carefully developed performance scale, and
3) a scientifically sound validation of both the construct and the scale.

The best known of the early tests used to predict programmer performance was the IBM Programmer Aptitude Test (PAT). This test contained three tasks which required job candidates to figure out the next number in a series, figure out analogies represented in figures, and solve arithmetic problems. These tasks were fine measures of mental abilities and could be used to select people for almost any white collar job in the company. Unfortunately, the relationships between this test battery and the job performance of programmers were often quite low (Reinstedt, 1966). Often these low correlations reflected little more than the well-known failure of managerial performance ratings to accurately represent individual performance. Even worse, experience in having taken the test improved subsequent scores (not a desired characteristic for a measure of native intellectual capacity). There were two reasons managers continued using tests of questionable validity:

1) they could shift responsibility for hiring decisions from themselves to the test, and
2) even with their weaknesses the tests were probably better judges of programming potential than were many managers.

Programmer selection testing had already fallen into disfavor by the Garmisch conference. This initial attempt of psychologists to aid software engineering had faired poorly not because the principles and technologies of psychology were not up to the task, but because the psychologists involved took the easy road out. Psychologists failed to adequately model the mental and behavioral aspects of programming before selecting tests to measure it.

Nevertheless, individual differences in performance among programmers remained a critical problem on programming projects. Sackman, Erickson, and Grant (1968) produced data displaying a 28:1 range in debugging performance. However, their data were confounded by the use of different programming languages. I subsequently reported debugging data collected with my colleagues at GE (Curtis,1981c) which displayed 23:1 differences without confounding factors. Boehm (1981) reported that differences in personnel and team capability was the most significant factor affecting programming productivity in his multi-year cost estimating study at TRW.

Recent efforts to develop more appropriate tests measuring individual differences among programmers have met with greater success. Wolfe (1971) developed a series of tests for assessing programming aptitude which require candidates to manipulate numbers according to an intricate set of procedures that are not unlike some assembler tasks. A validation study of one of these tests appears in DeNelsky and McKee (1974). The Wolfe tests for programming aptitude primarily assess an individual's ability to follow detailed procedural instructions. However, this skill is only one of those required of entry level programmers.

Ray Berger began with a thorough job analysis of programming jobs and subsequently produced a series of tests for assessing different levels of skill and knowledge in programming. His initial aptitude test requires candidates to learn a short procedural language and then use it in solving problems of increasing complexity. This is the only widely marketed test that directly assesses the ability of applicants to learn and use a language. This skill is especially important when hiring entry level programmers who will be placed in a training program. Studies of the Berger Aptitude for Programming Test (B-APT) have obtained some of the highest validities to date, although these studies have not been reported in the archival literature.

The bottom line after two decades of work on programmer selection is that the individual differences model has never been applied to programming as effectively as it should have been. Programmer selection research has rarely considered more than a few mental abilities. The full set of individual characteristics which affect programming performance has never been modelled and studied in the same set of data. Figure 1 presents some of the characteristics that would need to be considered in a model of individual programming performance. Most programmer selection tests only assess factors listed on the left side of Figure 1.

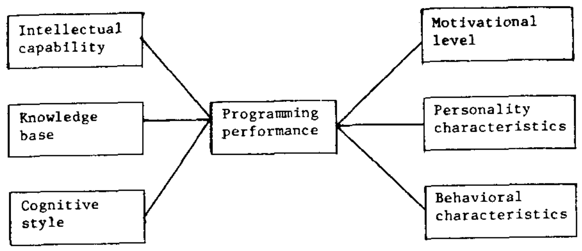

Figure 1. Factors affecting individual programming performance

Couger and Zawacki (1980) took the individual differences model further than most psychologists had in studying programmers. They identified how differences in the motivational structure of programmers interacted with the kinds of jobs they were assigned. They found that programmers had higher needs for personal growth and personal development than those in any other job category measured. However, programmers had lower needs for social interaction than people in most other types of jobs. This result should not be interpreted to imply that programmers are antisocial, rather that they get their greatest source of satisfaction from their job and their own professional development.

Couger and Zawacki used the Hackman and Oldham (1975) model of job characteristics to analyze programming jobs on the dimensions which have the greatest impact of the motivational structure of programmers. A summary of this analytic model is presented in Figure 2. The Hackman-Oldman model postulates that various characteristics of the job have substantial impact on the psychological state of the individuals performing the job. These job characteristics define its motivating potential. It is the job's motivating potential interacting with the primary motivations of the individual which will result in a level of performance, satisfaction, turnover, etc.

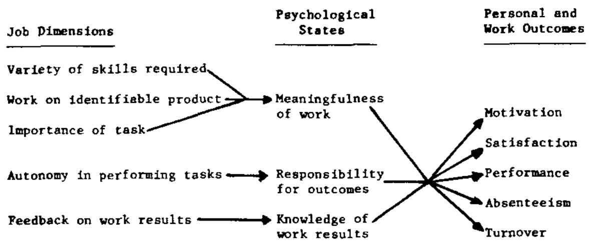
Job Dimensions
Psychological States
Personal and Work Outcomes
Variety of skills required
Work on identifiable product
Importance of task
Autonomy in performing tasks
Feedback on work results
Meaningfulness of work
Responsibility for outcomes
Knowledge of work results
Motivation
Satisfaction
Performance
Absenteeism
Turnover

Figure 2. Hackman-Oldham model of a job's motivating potential

Since programmers have shown such high levels of need for personal growth, it is important that their jobs be structured to provide high levels of the five job dimensions. Programmers who do not have the characteristic strong growth need may be more suited for programming jobs with less motivating potential. This model makes a sophisticated use of the individual differences approach by considering the match between personal characteristics and the surrounding environment.

Until the many sources of variation among individuals have been compared in the same set of data, it will not be possible to determine precisely which of the potential sources is the most important predictor of success in training programs or on the job. Further, as Weinberg (1971) suggested over a decade ago, it is unclear that we have assessed all of the important mental abilities related to programming. He specifically pointed toward a failure to assess the ability to consider alternate causal explanations of erroneous operation during debugging. More recently, Green (1977) has shown that the type of task involved in debugging is separate from the type involved in writing code.

It would appear from work in cognitive science (to be discussed later) that the most important determinant of individual differences in programming is the knowledge base possessed by a programmer. As will be described in the section on cognitive science, the performance of someone tackling a complicated programming task is related to the richness of their knowledge about the problem area. Thus, while the individual differences paradigm provides a method for predicting performance differences among programmers, it fails to offer an explanation of why these differences occur or how to reduce them. Although the individual differences paradigm attempts to assess the mental structure of a human being, it rarely captures the dynamic growth or interaction among these structures. Its limitation is that it presents a static model of human beings. A better model will be needed to explain how individual differences occur.

The following points summarize the almost three decades of research which have investigated the individual differences among programmers.

- It took two decades to realize that programmers had more than one dimension.
- Managers rely on aptitude tests like a drowning sailor grasping for floating debris.
- A test is no better than the job analysis and validation study which supports it.
- Good tests are currently available, but they should only constitute part of the selection process.
- Advances in individual difference models will come from a better understanding of the programmer knowledge base.

Cognitive Science

The paradigm of cognitive science seeks to understand how knowledge is developed, represented in memory, and used. The interaction of cognitive science and computer science has led to the emergence of artificial intelligence, the attempt to make computers process information in ways similar to those used by humans.

There are several different levels at which researchers have modelled cognitive processes in programming. The differences in the models presented here are primarily in the levels of explanation. Cognitive theories of programming have not been elaborated to the extent that they present alternative explanations of programmer performance. In fact, on the surface many of the theories are interesting applications of psychological principles to programming, but they have not been sufficiently elaborated for consistent practical application at a technical level. Nevertheless, the models presented here are promising approaches to understanding how programmers develop programs.

Most cognitive models of programming begin with the distinction between short and long term memory. Short term memory is a limited capacity workspace which holds and processes those items of information currently under our attention. The capacity of short term memory was originally characterized by Miller (1956) as holding 7 ± 2 items. An item is a single piece of information, although there is no requirement that it be an elementary piece resulting from the decomposition of a larger body of information.

Currently, many cognitive theorists portray short term memory as allocating the scarce resources of the cognitive processor, rather than as possessing a limited number of mental slots for information. Nevertheless, this limited capacity information buffer provides one of the greatest limitations to our ability to develop large scale computer systems. That is, we simply cannot think of enough things simultaneously to keep track of the interwoven pieces of a large system.

A process called 'chunking' expands the capacity of our short term mental workspace. In chunking, several items with similar or related attributes are bound together conceptually to form a unique item. For instance, through experience and training programmers are able to build increasingly larger chunks based on solution patterns which emerge frequently in solving problems. The lines of code in the program listing:

```
    SUM = 0
    DO 10 I = 1, N
    SUM = SUM + X(I)
 10 CONTINUE
```

would be fused by an experienced programmer into the chunk "calculate the sum of array X". The programmer can now think about working with an array sum, a single entity, rather than the six unique operators and seven unique operands in the four program statements above. When it is necessary to deal with the procedural implementation, the programmer can call these four statements from long term memory as underlying the chunk "array sum".

Much of a programmer's maturation involves observing more patterns and building larger chunks. The scope of the concepts that programmers have been able to build into chunks provides one indication of their programming ability. The particular elements chunked together have important implications for educating programmers. Educational materials 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. An important concern with long term memory is how the information stored there is interrelated and indexed such that:

1) items in short term memory can quickly cue the recall of appropriate chunks of information from long term memory,
2) items in short term memory can be linked into and transferred quickly to long term memory for retention, and
3) information retrieved from long term memory can cue the retrieval of additional chunks of information when appropriate.

The effects of both experience and education are on the knowledge base they construct in long term memory. The construction of this base is not merely one of accumulating facts, but of organizing them into a rich network of semantic material.

Shneiderman and Mayer (1979) have characterized the structure of knowledge in long term memory into a syntactic/semantic model. In their model, syntactic and semantic knowledge are organized separately in memory. 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 idiosyncracies of a given programming language.

An important implication of the Shneiderman and Mayer model is that the development of programming skill requires the integration of knowledge from several different knowledge domains (Brooks, 1983). For instance, the programming of an on-board aircraft guidance system may require knowledge of:

1) aeronautical engineering
2) radar and sensors technology
3) mathematical algorithms
4) the design of the on-board processor
5) the development machine and tools
6) a high level programming language
7) an assembly language

Each of these is a separate field of knowledge, some of which require years of training and experience to master. Thus, programming skill is specific to the application being considered. One can be a talented avionics programmer, and still be a novice at programming simultaneous multi-user business databases.

Several efforts have been made to model the structure of programming knowledge at a level deeper than that of Shneiderman and Mayer. Brooks (1977) used Newell and Simon's (1972) production system approach to model the rules a programmer would use in writing the code for a program. These rules are of the type, "If the following conditions are satisfied, then produce the following action". Based on analysis of a verbal protocol, Brooks identified 73 rules which were needed to model the coding process of a single, and relatively simple, problem solution. Brooks estimated that the number of production rules needed to model the performance of an expert programmer was in the tens to hundreds of thousands.

Atwood, Turner, Ramsey, and Hooper (1979) modelled a programmer's understanding of a program using Kintsch's (1974) model of text comprehension. Their approach treats a program as a text base composed of propositions. Comprehension occurs as elementary or micro-propositions are fused into macro-propositions which summarize their meaning or content. This process is similar to chunking. The result of this process is a hierarchy of macro-propositions built from the micro-propositions at the bottom of the tree. A micro-proposition is a simple statement composed of a relational operator and one or more arguments (operands).

Atwood et al. (1979) demonstrated that a program design could be broken into a hierarchical structure of propositions. They observed that after studying the design, more experienced programmers were able to recall propositions at a greater depth in the hierarchy than novices. The more experienced programmers had more elaborate structures in long term memory for use in encoding such designs. Thus, they were able to retain propositions at greater depth because:

1) the higher level macro-propositions in the design did not represent new information, and thus could be referenced by existing knowledge structures, and
2) the propositions representing new information could be linked into the existing knowledge structures of experienced programmers and shifted into long term memory.

This propositional hierarchy is one representation of how knowledge is structured in long term memory. To understand how these knowledge structures develop, cognitive scientists have studied differences between expert and novice programmers.

Expert-novice differences. The study of expert-novice differences in programming has generated information on how the programming knowledge base is developed. Both Adelson (1981) and Weiser and Shertz (1983) demonstrated that novices comprehend a program based on its surface structure, that is, the particular applications area of the program such as banking, or avionics. Experts, however, analyze a program based on its deep structure, the solution or algorithmic structure of the program. Similarly, McKeithan, Reitman, Rueter, and Hirtle (1981) observed that experts are able to remember language commands based on their position in the structure of the language. Novices, not having an adequate mental representation of the language structure, often use mnemonic tricks to remember command names.

Results of the expert-novice differences research in programming agree with the results of similar research on other subject areas (e.g., thermodynamics, physics, and chess) conducted by Herbert Simon and his associates at Carnegie-Mellon. They have determined that experts are not necessarily better at operational thinking than novices. Rather, experts are better at encoding new information than novices. The broader knowledge base of experts guides them to quickly cue in on the most important aspects of new information, analyze them, and relate them to appropriate schema in long term memory.

Developing technical skill is not merely a matter of learning a long list of facts. Rather, developing technical skill is an effort to learn the underlying structure of the knowledge required for the task. McKeithan et al. found that the knowledge structures developed by experts were more similar to each other than were those of intermediates or novices. Thus, programmers tend to gravitate toward a similar understanding of the language structure with experience. The development of this structure enhances the ability of experienced programmers to assimilate new information.

Soloway and his colleagues at Yale (Soloway, Bonar, and Ehrlich, 1983; Soloway, Ehrlich, and Bonar, 1981) have modelled the programming knowledge base as a collection of plans or templates. These plans represent the algorithmic or computational structures programmers use in conceiving the solution to a problem. These plans become more efficient and elaborate as programmers gain in experience. Soloway et al. (1983) demonstrated that programmers can work more effectively when the language they use supports the structure of the templates in their knowledge base.

In a psychological study of the program design process, Jefferies, Turner, Polson, and Atwood (1981) noted that programmers with greater experience decomposed a problem more richly into minimally interacting parts. The design knowledge of novices did not appear sufficient to provide for a full decomposition. In particular, more experienced programmers spent greater time evaluating the problem structure prior to beginning the design process. Observations similar to these were also made by Nichols (1981).

Jeffries et al. hypothesized that there is the equivalent of a mental design executive. This executive attempts to recursively decompose the problem statement and relate the components emerging from the decomposition to patterns in the programming knowledge base in long term memory. The shallowness of the novices' decomposition reflects the shallowness of the knowledge base against which they attempt to compare pieces of the problem statement. The richer knowledge base of experts allows them a fuller decomposition of the problem statement. The criterion used by experts for terminating the decomposition process for a particular aspect of the problem is when it has been decomposed to a level for which the programmer can retrieve a solution template.

Design problem solving. Most problem solving research has been performed on well defined problems with finite solution states. In problems such as the Towers of Hanoi, there is an optimal path to the solution. The path to a successful solution in chess is not so clearly defined. Nevertheless, in chess there are a finite number of moves which can be chosen at any time and a well defined solution state. In a semantically rich domain such as programming, neither are the options from which one can choose limited nor is there a clearly defined solution state. Therefore, studying problem solving in programming is a qualitatively different task than most of those used in problem solving research.

Carroll, Thomas, and Malhotra (1980) argued that solving unstructured problems could not be explained with existing theory. They began their investigations of the design process by studying how analysts and clients interacted in establishing the requirements for a system. Carroll, Thomas, and Malhotra (1979) observed that client/analyst requirements sessions were broken into cycles which represented the decomposition of the problem. However, these cycles did not decompose the problem in a top-down fashion as recommended by structured programming practices. Rather, these cycles represented a linear or sequential decomposition of the problem in which the sub-problem to be attacked in the next cycle was cued by the results of the last cycle. The only a priori structure placed on the content of these client/analyst cycles was determined by the initial goal structure of the client.

The problem in moving from the idea for a system to its final implementation is in transforming a linearly derived sequence of desired components into a hierarchical arrangement of functions or data transformations. Once the requirements have been delineated, they must be organized so that the inherent structure of the problem becomes visible. The next step is to construct a solution structure which matches the problem structure. To the extent that these structures are logically organized and matched, the system will possess a structural integrity which can expedite its implementation.

A series of studies by Carroll and his associates at IBM's Watson Research Center identified several factors which impact the effectiveness of designing a solution. First, they demonstrated differences in problem analysis based on differences in the application attempted. It has been consistently found in problem solving research that people do not transfer solution structures across problem isomorphs. Isomorphs are problems with the same structural characteristics, but whose cover stories (or subject areas) differ. Previous problem solving research has established that there is poor transfer of previously learned problem solutions across isomorphs. The structure of the cover story affects the difficulty people experience in reaching a solution.

Carroll et al. (1980) observed that people had more difficulty solving a problem that involved temporal relations (designing a manufacturing process) than an isomorph which involved spatial relations (arranging an office layout). The difference arose in part because the spatial problem lends itself to graphical representation. However, the temporal isomorph does not present spatial cues and participants had difficulty representing it to themselves. Many retreated to a verbal description of the problem, and several were totally unable to solve it. When a graphical aid was provided for solving the temporal problem, it appeared to make the problem easier to understand. The spatial aid did not make the problem easier to solve, however, since the same number of participants were unable to solve it.

The structuring of the requirements also seems to have an impact on the characteristics of the problem solution. Presenting the requirements in clusters based on their inherent structure assisted participants in designing solutions which better reflected the problem structure and were more stable when new requirements were added (Carroll, Thomas, Miller, & Friedman, 1980). Greater structure in the original problem statement seems to reduce the amount of iteration through design cycles. Thus, a critically important focus of the structured programming movement should be on methods of structuring the statement of requirements. Far less attention has been paid this problem than to areas, such coding, that have less impact on system integrity and costs.

Hoc (1981) studied the results of designing a program from the data structure versus the results structure. He suggested that a choice of design method is often made prematurely, prior to understanding the relation between the data and results structures and the processing which transformed the former into the latter. He felt that the choice of design method was better made after this problem analysis stage.

The conceptual integrity of the program design is critical to the success of a programming project. No level of management talent can sustain high productivity and quality on a project which fails to achieve it. A most critical area for programming research, then, is requirements and design techniques. The current level of behavioral research on these topics is only a start in what needs to be a major thrust.

<u>The detection of procedural faults</u>. At least part of the process of developing an organized knowledge structure about programming is the abstraction of rules from the myriad patterns and facts that programmers know or can recognize. Whereas an expert programmer may be able to recognize 50,000 patterns, the number of rules which govern the structure of these patterns is substantially less, perhaps 1000 to 3000. Brooks (1977) estimated many more rules, but he may have been referring to the recognizable patterns from which these rules are drawn. Rule-based knowledge in programming has been studied most frequently in the detection of procedural faults.

One of the most critical and time consuming tasks in programming is the detection and correction of faults (bugs). While debugging has been used as an experimental task for studies on specifications or language features, relatively little behavioral research has been directed toward understanding the debugging process.

John Seely Brown and his associates (Brown & Burton, 1978; Brown & Van Lehn, 1980) have laid some theoretical groundwork for modelling the generation of bugs in procedural tasks. They treat bugs not as random occurances, but as systematic and predictable outcomes of the incomplete or incorrect application of the rules underlying a procedural skill. Their explanation entails four components:

1) the first component is that an individual acquire a formal representation of a procedural skill. Such a representation would be a set of rules which guide the development of procedures for solving a problem.
2) the second component of their model is a set of principles for determining which rules can be deleted from the formal representation to simulate the incomplete or incorrect learning of rules or the forgetting of rules.
3) the third component is a set of repair heuristics used by the individual to patch over gaps in the formal representation. These heuristics generate bugs by creating inappropriate procedures for completing procedural solutions.
4) the final component is a set of mechanisms for screening out some of the heuristics which generate blatently incorrect procedures.

This type of model is guiding some of the current work on intelligent debugging aids (Johnson & Soloway, 1984).

Youngs (1974) reported some descriptive data on the types of errors typically made by programmers. His data were similar to several databases collected on large system development projects by the Information Sciences Program at Rome Air Development Center. The most frequent category of faults was logic errors, especially for experienced programmers. Syntactic errors occurred relatively infrequently. This observation reinforces the importance of providing useful control constructs in the programming languages. The results also indicate that novices and professionals make different kinds of errors.

During the early 1970s John Gould and his associates at IBM's Watson Research Center made several studies of program debugging. In the first study, Gould and Drongowski (1974) found that providing debugging aids to programmers did not necessarily make fault detection faster. Programmers adopted debugging strategies based on the types of information they were presented about the program and the problem. This strategy included attempts to localize the section of code likely to contain the error, and employed a hierarchical search in which the most complex sections were left for last. In a further study, Gould (1975) identified that this hierarchical search was for:

1) syntactic faults,
2) grammatical faults not caught by the compiler, and
3) substantive faults.

Sheppard, Curtis, Milliman, and Love (1979) observed several different search strategies among programmers. Some programmers felt they had to understand the entire program before they could begin searching for the fault. The more effective strategy, however, was to identify that portion of the output which was in error and quickly trace back from the print statement for that variable to locate the area in which the fault was likely to have occurred. This technique is similar to the program slicing strategy studied by Weiser (1982).

One of the most extensive programs of research on fault diagnosis has been conducted by William and Sandra Rouse now at Georgia Tech. They have made an important distinction between perceptual complexity and problem solving complexity (Rouse & Rouse, 1979). They suggest that the latter is more affected by individual differences, especially those related to understanding a problem. Brooke and Duncan (1981) demonstrated that factors which primarily impact perceptual complexity, such as the display format, can affect problem solving effectiveness. Subsequently, Rouse, Rouse, and Pelligrino (1980) have developed a rule-based model of fault diagnosis that agrees at a global level with the actual performance of people on a similar task.

Learning to program. There are two primary ways in which the rules which govern programming can be learned. They can be abstracted from the developing knowledge base as the programmer gains increasing experience. This, of course, is a lengthy process. On the other hand, rules can be taught in organized training programs. Training not only develops the knowledge base more quickly than experiential learning, but it is also likely to be more thorough and accurate. However, experiential learning is often the primary method for acquiring the contextual information used in interpreting the appropriateness of various rules for programming.

Mayer (1976, 1981) described several training techniques grounded in psychological theory and research which can be used successfully in training novice programmers. Mayer (1976) stressed the importance of 'advanced organizers' to help structure new material as it is learned. These advanced organizers help build a preliminary model or outline of the new information so that later input can be more easily assimilated into an appropriate knowledge structure. Mayer emphasized that one of the most effective advanced organizers is a concrete model of the machine which is manipulated by instructions coded in a computer language. Mayer argued that students benefit from being forced to elaborate these models in their own words.

DuBoulay, O'Shea, and Monk (1981) extended Mayer's concept of a concrete model of the machine. They discussed a 'notional machine' which is a simplified machine whose facilities are only those which are implemented by the available commands in the programming language. They also stressed the importance of a student's gaining visibility into the processes occurring inside the abstract notional machine. They have built several training systems based on this concept.

Coombs, Gibson, and Alty (1982) have identified two learning styles which characterize the different ways novices learn to program: comprehension learning and operational learning. Comprehension learners acquire an overall layout of the information under study, but may not understand the rules which allow them to operate with and on the information. Operational learners grasp the rules for operating on information, but they do not acquire a complete picture of the knowledge domain. Comprehension learners are primarily interested in understanding, while operational learners are primarily interested in doing something. These characterizations represent idealized students, whereas most people will fall on a continuum in between, displaying varying degrees of both styles.

Coombs et al. concluded from their data that operational learners were better able to learn a programming language. Their learning strategy was characterized by attention to the details of the language structures, the abstraction of critical language features, and an orientation towards representing important structural relations in rules. The major learning activity for operational learners was in practice sessions, whereas for comprehension learners this occurred in lectures.

Lemos (1979) investigated the benefits of structured walkthroughs as a classroom learning exercise. This approach seemed to have advantages in allowing novices to compare alternative approaches to the problem and gain immediate feedback on their strategy. Shneiderman (1980) has used a similar feedback mechanism with experienced programmers and found benefits in terms of learning new approaches to a problem.

Conclusions

I have argued that individual differences among project personnel (and this should be even more true in the unstudied area of programming managers) accounts for the largest source of variation in project performance. In fact, Sheppard, Kruesi, and Curtis (1981) found that half of the variation in the efficiency of extracting information from different documentation formats was attributable to individual differences among the professional programmers involved in the experiment. However, the individual differences paradigm only allows us to characterize and predict these differences, but not explain how they develop and change over time. Cognitive science has provided a representation of knowledge organization and development which presents an explanation of the basis for these differences. Therefore, cognitive science is a paradigm which offers the best opportunity to study and gain control over the largest source of influence of project performance.

Cognitive science presents an opportunity for psychologists to get on the leading edge of programming technology, rather than sweeping up behind the directions already set by computer scientists. As a driving force in artificial intelligence, cognitive science provides a vehicle for analyzing the most appropriate ways to automate more of the programming process in ways that are helpful to those who must develop large systems.

The following points describe some of the important themes emerging from cognitive science research on software engineering:

- Expertise is specific to different knowledge domains. A programmer can be expert in one domain and a novice in another.
- The development of expertise involves building a massive knowledge base of recognizable patterns (perhaps 50,000) and abstracting a set of rules (perhaps 1000 to 3000) which govern their behavior.
- Rule-based models of programming need to be expanded far beyond their current use, primarily in fault diagnosis. Rule-based models hold substantial promise for automating programming tasks.
- So little research is being performed on the problem solving process during requirements definition, functional specification, and program design, that this must be a crucial area for improving software engineering practice.
- Rather than teaching isolated commands, educators should liberally model abstract machines for teaching the structure of a programming language.
- Learning styles will play an important role in how quickly, accurately, and thoroughly an individual learns to program.
- Cognitive science is the easiest way for a psychologist to communicate with a computer scientist, but someone with artificial intelligence may have to interpret one to the other.

There will remain behavioral questions with significant impact on the usefulness of new developments in programming technology. Some of these questions involve:

1) techniques for insuring the completeness of a requirements statement,
2) techniques for clustering the requirements to better reveal the inherent structure of the problem,
3) techniques for deciding on the allocation of requirements between hardware and software,
4) techniques for bridging the gap between a statement of requirements and the preliminary program design,
5) techniques for indexing and retrieving reused program modules,
6) techniques for proving the correctness of reused program modules,
7) techniques for coordinating the work of project team members, and
8) techniques for designing and verifying the data flow among modules.

Thus, there will need to be a shift in emphasis in behavioral research away from coding issues toward the concerns enumerated above. If behavioral scientists, and especially psychologists, begin attacking these problems immediately, they can influence the development of new technology in software engineering. Further, the models of programmer performance being developed by cognitive scientists can be useful in developing knowledge-based tools and environments for software engineering.

REFERENCES

Adelson, B. Problem solving and the development of abstract categories in programming languages. Memory and Cognition, 1981, 9(4), 422-433.

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

Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Brooke, J.B. & Duncan, K.D. Effects of system display format on performance in a fault location task. Ergonomics, 1981, 24(3), 175-189.

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.

Brooks, R. Towards a theoretical model of the comprehension of computer programs. International Journal of Man-Machine Studies, 1983, 17.

Brown, J.S. & Burton, R.R. Diagnostic models for procedural bugs in basic mathematics skills. Cognitive Science, 1978, 2, 155-192.

Brown, J.S. & VanLehn, K. Repair theory: A generative theory of bugs in procedural skills. Cognitive Science, 1980, 4, 379-426.

Carroll, J.M., Thomas, J.C., & Malhotra, A. Clinical-experimental analysis of design problem solving. Design Studies, 1979, 1(2), 84-92.

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

Carroll, J.M., Thomas, J.C., Miller, L.A., & Friedman, H.P. Aspects of solution structure in design problem solving. American Journal of Psychology, 1980, 93(2), 269-284.

Coombs, M.J., Gibson, R., & Alty, J.L. Learning a first computer language: strategies for making sense. International Journal of Man-Machine Studies, 1982, 16, 449-486.

Couger, J.D. & Zawacki, R.A. Motivating and Managing Computer Personnel. New York: Wiley, 1980.

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

Curtis, B. A review of human factors research on programming languages and specifications. Proceedings of Human Factors in Computer Systems. New York: ACM, 1981. b

Curtis, B. Substantiating programmer variability. Proceedings of the IEEE, 1981, 69(7), 846. c

DeNelsky, G.Y. & McKee, M.G. Prediction of computer programmer training and job performance using the AABP test. Personnel Psychology, 1974, 27, 129-137.

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

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

Gould, J.D. & Drongowski, P. An exploratory study of computer program debugging. Human Factors, 1974, 16(3), 258-277.

Hackman, J.R. & Oldham, G.R. Development of the job diagnostic survey. Journal of Applied Psychology, 1975, 60(2), 159-170.

Hoc, J.M. Planning and direction of problem solving in structured programming: An empirical comparison between two methods. International Journal of Man-Machine Studies, 1981, 15, 363-383.

Jefferies, R., Turner, A.A., Polson, P.G., & Atwood, M.E. The processes involved in designing software. In J.R. Anderson (Ed.), Cognitive Skills and Their Acquisition. Hillsdale, NJ: Erlbaum, 1981, 255-283.

Johnson, W.L. & Soloway, E. PROUST: Knowledge based program understanding. Proceedings of the Seventh International Conference on Software Engineering. Silver Spring, MD: IEEE, 1984.

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

Lemos, R.S. An implementation of structured walkthroughs in teaching Cobol programming. Communications of the ACM, 1979, 22(6), 335-340.

Mayer, R.E. Some conditions for meaningful learning in computer programming: Advance organizers and subject control of frame order. Journal of Educational Psychology, 1976, 68, 143-150.

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, 307-325.

Miller, G.A. The magical number seven plus or minus two: Some limits on our capacity to process information. Psychological Review, 1956, 63, 81-97.

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.

Reinstedt, R.N. et al. Computer personnel research group programmer performance prediction study. Proceedings of the Fifth Annual Computer Personnel Research Conference. New York: ACM, 1966.

Rouse, W.B. & Rouse, S.H. Measures of complexity of fault diagnosis tasks. IEEE Transactions on Systems, Man, & Cybernetics, 1979, 9(11), 720-727.

Rouse, W.B., Rouse, S.H., & Pelligrino, S.J. A rule-based model of human problem solving performance in fault diagnosis tasks. IEEE Transactions on Systems, Man, and Cybernetics, 1980, 10(7), 366-376.

Sackman, H., Erickson, W.J., & Grant, E.E. Exploratory and experimental studies comparing on-line and off-line programming performance. Communications of the ACM, 1968, 11, 3-11.

Sheil, B.A. The psychological study of programming. ACM Computing Surveys, 1981, 13(1), 101-120.

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

Sime, M.E., Green, T.R.G., & Guest, D.J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.

Soloway; E., Bonar, J., & Ehrlich, K. Cognitive strategies and looping constructs: An empirical study. Communications of the ACM, 1983, 26(11), 853-860.

Soloway, E., Ehrlich, K., & Bonar, J. Tapping into tacit programming knowledge. Proceedings of Human Factors in Computer Systems. New York: ACM, 1982, 52-57.

Weinberg, G.M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.

Weiser, M. & Shertz, J. A study of programming problem representation in novice programmers. International Journal of Man-Machine Studies, 1983, 17.

Weiser, M. Programmers use slices when debugging. Communications of the ACM, 1982, 25(7), 446-452.

Wolfe, J. Perspectives on testing for programmer aptitude. Proceedings of the 1971 Annual Conference of the ACM. New York: ACM, 1971, 268-277.

Youngs, E.A. Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.

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