Expertise in debugging computer programs: A process analysis

IRIS VESSEY

Department of Commerce, University of Queensland, St Lucia, Brisbane, Australia 4067

(Received 10 March 1985)

This paper reports the results of an exploratory study that investigated expert and novice debugging processes with the aim of contributing to a general theory of programming expertise. The method used was verbal protocol analysis. Data was collected from 16 programmers employed by the same organization. First, an expert-novice classification of subjects was derived from information based on subjects' problem solving processes: the criterion of expertise was the subjects' ability to chunk effectively the program they were required to debug. Then, significant differences in subjects' approaches to debugging were used to characterize programmers' debugging strategies. Comparisons of these strategies with the expert-novice classification showed programmer expertise based on chunking ability to be strongly related to debugging strategy. The following strategic propositions were identified for further testing. 1. (a) Experts use breadth-first approaches to debugging and, at the same time, adopt a system view of the problem area; (b) Experts are proficient at chunking programs and hence display smooth-flowing approaches to debugging. 2. (a) Novices use breadth-first approaches to debugging but are deficient in their ability to think in system terms; (b) Novices use depth-first approaches to debugging; (c) Novices are less proficient at chunking programs and hence display erratic approaches to debugging.

1. Introduction

The spate of recent literature on the cognitive psychology of programming attests to the growing interest in determining the cognitive principles underlying computer programming (e.g. see reviews by Shneiderman, 1980; Smith & Green, 1980; Sheil, 1981; Pennington, 1982). The study of programming processes is important for two reasons. Firstly, researchers must control for the knowledge structures that programmers possess if they wish to measure the *effects* of factors that influence programmer performance, namely, factors such as language design, program layout, programming mode and programming support facilities. Secondly, understanding the knowledge structures that expert and novice programmers possess is important *per se*: research at this level will contribute to a general theory of expertise in programming. It will therefore aid in such tasks as the design of programming languages, programming aids, programmer rating instruments, and programmer recruitment and training procedures.

This study investigated debugging processes with the aim of contributing to a general theory of programmer expertise.† Specifically, it sought to determine differences in the

†Debugging is the process of locating and correcting the error within the program. It differs from the related activity of testing in that testing reveals the presence of errors; hence, debugging follows testing (Myers, 1978).

0020-7373/85/110459+36$03.00/0

debugging processes of expert and novice programmers from the community of programming professionals. Since it was essential to capture what occurred during problem solving rather than merely the outcome of problem solving, the process tracing technique of recording verbal protocol was used as the method of data collection. Sixteen subjects, eight of whom were classed as experts and eight as novices, debugged a COBOL program, speaking aloud as they did so. This trace of their problem-solving was tape-recorded, transcribed, and then analyzed.

The following section (section 2) presents the basic philosophy underlying this investigation of debugging processes. Section 3 describes the research approach used in the study—it introduces the task materials, presents three tools for describing problem-solving processes, and describes the programmer classification methods tested in this research. The fourth section assesses the classification methods and selects one for further analysis. It then presents the results of analyzing subjects' debugging processes. The fifth section discusses the implications of the results for debugging processes and for the concept of programmer expertise, while the sixth discusses the limitations of the research. The paper concludes with the contributions the study makes to a theory of programming expertise and hence provides directions for future research in the area.

2. Conceptual approach to studying expertise

Historically, interest in the field of computer programming focused first on the development of programmer rating instruments, and then on factors that influence the programming process. The major outcome of the research into programmer assessment was the recognition that instruments frequently captured those variables that related to success in training courses but not those that related to performance on the job (Mayer & Stalnaker, 1968). Despite this evidence of the complex nature of expert programming skill, researchers in computer science embarked on numerous studies that attempted to measure the effects of various programming factors on the ease of programming. Not surprisingly, the results of those studies were mixed (Sheil, 1981; Pennington, 1982). Frequently, the variability among programmers was greater than between the levels of the experimental variables, suggesting yet again the need to control for some element of programmer skill.

Many researchers now believe that the uncontrolled variable is the process or knowledge structures programmers employ during problem solving (Brooks, 1980; Sheil, 1981; Vessey & Weber, 1984). Knowledge structures are cognitive units that accumulate in long-term memory as a result of experience (Newell & Simon, 1972). As programmers are exposed to a greater variety of programming situations, both the number and complexity of knowledge structures in long-term memory increase. Brooks (1977) suggests that a typical programmer's knowledge base may consist of 50 000 chunks. Hence, the resources potentially available to a programmer in solving a problem are many and varied. They may well affect a particular programming task to a greater extent than, say, indentation or the use of flowcharts, and thus lead to the mixed results of programming practices research. In the same way, the current investigation of expert and novice debugging processes could also suffer from a clear definition of expert and novice programmers, resulting in yet another study producing inconclusive results.

To address the problem of the variability in programmers' debugging processes, this study used two methods of classifying subjects. The first was the traditional *ex ante*

method of manager assessment. The second was an *ex post* process approach based on certain differences in subjects' problem solving processes. The two methods were then compared to determine the effectiveness of the process approach in reducing the variability in programmer performance.

2.1. CONTROLLING FOR DEBUGGING PROCESSES

The method used to control for differences in problem-solving processes was based on the efficiency of debugging processes. The criterion used was the subjects' ability to chunk programs: the more expert the programmers, the greater will be their chunking ability. The chunking ability of programmers was measured relative to a model of debugging functions (Fig. 1). Debugging functions are gross states of behaviour that programmers exhibit in debugging computer programs. The model shows those behaviours and the interrelationships between them.†

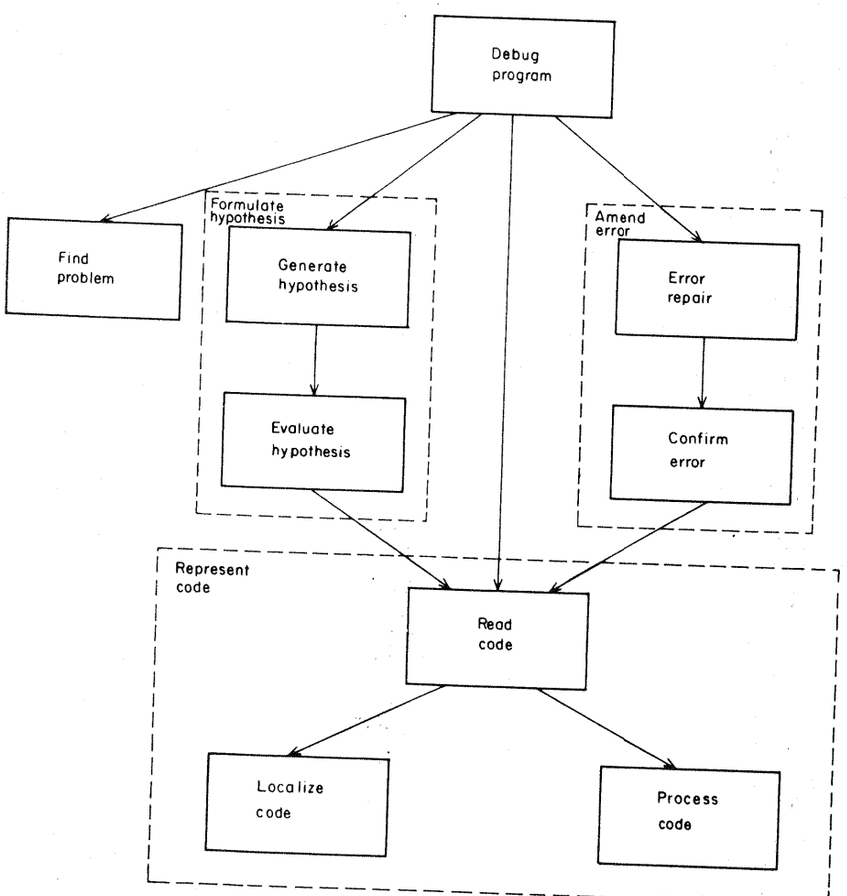

FIG. 1. Model of debugging functions.

† Vessey (1984, Table 3.1) shows the literature supporting inclusion of each function represented in Fig. 1.

Experts will demonstrate chunking ability by displaying a smooth approach to problem-solving. There will be little need to return to previous debugging functions or to parts of the program they have already seen. Novices, on the other hand, are expected to exhibit much more erratic behavior by rechecking clues and by returning to parts of the program they have already inspected. The ability to chunk during debugging can be characterized by three debugging efficiency criteria:

(1) the adoption of different debugging functions;

(2) reversion to the top or controlling Debug Program function to check again on the problem;

(3) change of location within the program.

The program's DATA DIVISION, modules of the PROCEDURE DIVISION, and the input and output listings are regarded as "program locations" for the purposes of this research. Compared to experts, therefore, it is expected that novice programmers will exhibit more changes in problem-solving functions, more reversals to the Debug Program function, and more changes of location in the material supplied.

2.2. ASSESSING THE EFFECTIVENESS OF THE RESULTING PROGRAMMER CLASSIFICATION

Since this method of programmer classification was derived directly from the research data, it was essential to have a means of assessing its effectiveness in distinguishing programmer skills. This was achieved in this study by comparing the effects of the manager and the *ex post* classifications on two objective performance criteria. The debugging effectiveness criteria chosen were:

(1) debug time;

(2) the number of errors subjects made.

If this method of classification were to succeed in reducing the variability in these objective performance factors relative to the manager classification, it would demonstrate the importance of controlling for problem-solving processes in programming research. Further, it would lead to better groupings of expert and novice programmers in this study and would therefore increase the possibility of deriving meaningful results from the other analyses performed.

3. Research method

The use of a process tracing technique is central to the investigation of problem-solving processes, i.e. a technique that captures what happens *during* problem-solving rather than merely the *outcome* of problem-solving. Process tracing methods include recording verbal protocol, monitoring information acquisition, and monitoring eye movements (Payne, Braunstein & Carroll, 1978). The first of these, recording verbal protocol, was chosen for use in this study since it results in much more data than the other two approaches; also the latter two methods demonstrate that problem-solvers reference data but not that they necessarily use it in problem-solving. That verbal protocol recording is the preferred method for examining problem-solving processes currently available, is demonstrated by the number of studies that have used it. Following the pioneering work of Newell and Simon (1972) in cryptarithmetic, it has been used in a variety of domains: physics (Simon & Simon, 1978; Larkin, McDermott, Simon & Simon, 1980; Larkin, 1981; Chi, Feltovich & Glaser, 1980), mathematics (Anderson, Greeno, Kline & Neves, 1981; Lewis, 1981), financial analysis (Bouwman, 1978, 1983; Biggs, 1978a, b), software design (Malhotra, Thomas, Carroll & Miller, 1980; Jeffries, Turner & Polson, 1980), and systems analysis (Vitalari, 1981; Vitalari & Dickson, 1983).


```
0217
0218          PROCEDURE DIVISION.
0219
0220    ****************************************************************
0221    *                                                              *
0222    *   THIS MODULE INITIALIZES THE FILES AND THEN DETERMINES WHEN  *
0223    *   CONTROL BREAKS HAVE OCCURRED AND CAUSES THE APPROPRIATE     *
0224    *   PROCESSING TO OCCUR. IT ALSO CAUSES THE DETAIL LINES TO     *
0225    *   BE PRINTED. IT IS ENTERED FROM THE OPERATING SYSTEM AND     *
0226    *   EXITS TO THE OPERATING SYSTEM                               *
0227    *                                                              *
0228    ****************************************************************
0229
0230     A000-CREATE-SALES-REPORT.
0231
0232         OPEN INPUT  SALES-INPUT-FILE
0233              OUTPUT SALES-REPORT-FILE.
0234         READ SALES-INPUT-FILE
0235               AT END
0236                  MOVE 'NO ' TO MORE-RECORDS.
0237         IF MORE-RECORDS EQUALS 'YES'
0238            MOVE CUSTOMER-NO-INPUT TO PREVIOUS-CUSTOMER-NUMBER
0239            MOVE SALESMAN-NO-INPUT TO PREVIOUS-SALESMAN-NUMBER
0240            MOVE BRANCH-NO-INPUT TO PREVIOUS-BRANCH-NUMBER
0241            PERFORM A001-PROCESS-AND-READ
0242                  UNTIL MORE-RECORDS EQUALS 'NO'
0243            PERFORM B010-PROCESS-CUSTOMER-CHANGE
0244            PERFORM B020-PROCESS-SALESMAN-CHANGE
0245            PERFORM B030-PROCESS-BRANCH-CHANGE
0246            PERFORM B040-PRINT-FINAL-TOTAL.
0247         CLOSE SALES-INPUT-FILE
0248               SALES-REPORT-FILE.
0249         STOP RUN.
0250
0251
0252
0253     A001-PROCESS-AND-READ.
0254
0255         IF BRANCH-NO-INPUT NOT = PREVIOUS-BRANCH-NUMBER
0256            PERFORM B010-PROCESS-CUSTOMER-CHANGE
0257            PERFORM B020-PROCESS-SALESMAN-CHANGE
0258            PERFORM B030-PROCESS-BRANCH-CHANGE
0259            MOVE 'YES' TO BRANCH-CHANGE
0260         ELSE
0261            IF SALESMAN-NO-INPUT NOT = PREVIOUS-SALESMAN-NUMBER
0262               PERFORM B010-PROCESS-CUSTOMER-CHANGE
0263               PERFORM B020-PROCESS-SALESMAN-CHANGE
0264               MOVE 'YES' TO SALESMAN-CHANGE
0265            ELSE
0266               IF CUSTOMER-NO-INPUT NOT = PREVIOUS-CUSTOMER-NUMBER
0267                  PERFORM B010-PROCESS-CUSTOMER-CHANGE
0268                  MOVE 'YES' TO CUSTOMER-CHANGE.
0269         PERFORM B000-PROCESS-DETAIL-RECORDS.
0270         READ SALES-INPUT-FILE
0271               AT END
0272                  MOVE 'NO' TO MORE-RECORDS.
0273
```

FIG. 2. Principal modules of the task program.

```
0274
0275
0276  ****************************************************************
0277  *                                                              *
0278  *   THIS MODULE IS ENTERED TO PRINT THE DETAIL LINE FOR THE    *
0279  *   REPORT.  IF NECESSARY, IT CAUSES THE HEADINGS TO BE PRINTED *
0280  *   AND THEN FORMATS AND PRINTS THE DETAIL LINE. TOTALS ARE ALSO *
0281  *   ACCUMULATED.  THIS MODULE IS ENTERED FROM THE              *
0282  *   A001-PROCESS-AND-READ MODULE AND EXITS BACK TO IT.         *
0283  *                                                              *
0284  ****************************************************************
0285
0286    B000-PROCESS-DETAIL-RECORDS.
0287
0288        IF LINES-PRINTED IS EQUAL TO PAGE-SIZE OR
0289              IS GREATER THAN PAGE-SIZE OR
0290              FIRST-PAGE
0291           PERFORM C000-PRINT-HEADINGS
0292           MOVE PREVIOUS-BRANCH-NUMBER TO BRANCH-NO-REPORT
0293           MOVE PREVIOUS-SALESMAN-NUMBER TO SALESMAN-NO-REPORT
0294           MOVE PREVIOUS-CUSTOMER-NUMBER TO CUSTOMER-NO-REPORT.
0295        IF BRANCH-CHANGE EQUALS 'YES'
0296           MOVE BRANCH-NO-INPUT TO BRANCH-NO-REPORT
0297           MOVE SALESMAN-NO-INPUT TO SALESMAN-NO-REPORT
0298           MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0299           MOVE 'NO' TO BRANCH-CHANGE
0300        ELSE
0301           IF SALESMAN-CHANGE EQUALS 'YES'
0302               MOVE SALESMAN-NO-INPUT TO SALESMAN-NO-REPORT
0303               MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0304               MOVE 'NO' TO SALESMAN-CHANGE
0305           ELSE
0306              IF CUSTOMER-CHANGE EQUALS 'YES'
0307                  MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0308                  MOVE 'NO' TO CUSTOMER-CHANGE.
0309        MOVE DESCRIPTION-INPUT TO DESCRIPTION-REPORT.
0310        MOVE SALES-INPUT TO SALES-REPORT.
0311        ADD SALES-INPUT TO CUSTOMER-TOTAL-ACCUM
0312                           SALESMAN-TOTAL-ACCUM
0313                           BRANCH-TOTAL-ACCUM
0314                           FINAL-TOTAL-ACCUM.
0315        WRITE SALES-REPORT-LINE FROM DETAIL-LINE
0316              AFTER PROPER-SPACING.
0317        ADD PROPER-SPACING TO LINES-PRINTED.
0318        MOVE 1 TO PROPER-SPACING.
0319        MOVE SPACES TO DETAIL-LINE.
0320
0321
0322
0323
0324
0325  ****************************************************************
0326  *                                                              *
0327  *   THIS MODULE IS ENTERED TO PROCESS A CHANGE IN CUSTOMER     *
0328  *   COMPARE AREA AND COUNTER.  IT IS ENTERED FROM THE          *
0329  *   A001-PROCESS-AND-READ MODULE AND ON COMPLETION FROM THE    *
0330  *   A000-CREATE-SALES-REPORT MODULE.                           *
0331  *                                                              *
0332  ****************************************************************
0333
0334    B010-PROCESS-CUSTOMER-CHANGE.
0335
0336        MOVE CUSTOMER-TOTAL-ACCUM TO CUSTOMER-TOTAL-CUSTOT.
```

FIG. 2. Continued.


```
0337        PERFORM B011-PROCESS-CUSTOMER-DISCOUNT.
0338        MOVE CUSTOMER-DISC-ACCUM TO CUSTOMER-TOTAL-DISTOT.
0339        WRITE SALES-REPORT-LINE FROM CUSTOMER-TOTAL-LINE
0340              AFTER ADVANCING 2 LINES.
0341        MOVE ZEROS TO CUSTOMER-TOTAL-ACCUM.
0342        ADD CUSTOMER-DISC-ACCUM TO SALESMAN-DISC-ACCUM.
0343        MOVE ZEROS TO CUSTOMER-DISC-ACCUM.
0344        MOVE CUSTOMER-NO-INPUT TO PREVIOUS-CUSTOMER-NUMBER.
0345        ADD 2 TO LINES-PRINTED.
0346        MOVE 2 TO PROPER-SPACING.
0347
0348
```

FIG. 2. Continued.

3.1. TASK

The program used was a straightforward COBOL sales reporting program with control breaks on branch number, salesperson number, and customer number. A simple application domain was used so that differences in application domain knowledge would not be a variable in the study. This permitted the investigation of debugging expertise alone. The program was fully structured. Figure 2 shows the first four modules of the program source code, while Fig. 3a shows the correct program output.

The error introduced was a logic error, a type commonly found in practice (Youngs, 1974; Gould & Drongowski, 1974; Gould, 1975; Sheppard, Curtis, Milliman & Love, 1979). No syntactic errors were present. As a basis for determining whether the task was sufficiently difficult to differentiate between experts and novices, the "same" bug was introduced at different locations in the program. Atwood and Ramsey (1978) report that an error both lower in the propositional hierarchy and lower in the program structure is more difficult to detect and correct than a similar error higher in the program structure.† Two versions of the program were produced with one error in each version. The module changed in the study is B000-PROCESS-DETAIL-RECORDS (see Fig. 2). The correct program logic is as follows:

```
0295        IF BRANCH-CHANGE EQUALS 'YES'
0296           MOVE BRANCH-NO-INPUT TO BRANCH-NO-REPORT
0297           MOVE SALESMAN-NO-INPUT TO SALESMAN-NO-REPORT
0298           MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0299           MOVE 'NO' TO BRANCH-CHANGE
0300        ELSE
0301           IF SALESMAN-CHANGE EQUALS 'YES'
0302              MOVE SALESMAN-NO-INPUT TO SALESMAN-NO-REPORT
0303              MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0304              MOVE 'NO' TO SALESMAN-CHANGE
0305           ELSE
0306              IF CUSTOMER-CHANGE EQUALS 'YES'
0307                 MOVE CUSTOMER-NO-INPUT TO CUSTOMER-NO-REPORT
0308                 MOVE 'NO' TO CUSTOMER-CHANGE.
```

† The term "propositional hierarchy" refers to the embedding or nesting of clauses in a sentence structure (Kintsch & van Dijk, 1978).

15/10/81 SALES REPORT PAGE 1

BRANCH NO	SALESMAN NO	CUSTOMER NO	PRODUCT DESCRIPTION	SALES AMOUNT	DISCOUNTED AMOUNT
100	1225	32911	AUDIO INTERFACE	500.00	
			KEYBOARD	100.00	
			POWER SUPPLY	50.00	
				650.00*	585.00*
		40015	CRT INTERFACE	75.00	
			FLOPPY CONTROLLER	125.00	
			POWER TRANSFORMER	50.00	
				250.00*	250.00*
			TOTAL SALESMAN NO 1225	900.00**	835.00**
	4199	24151	4K RAM	330.00	
			ROM MEMORY	30.00	
				360.00*	342.00*
			TOTAL SALESMAN NO 4199	360.00**	342.00**
			TOTAL BRANCH NO 100	1,260.00***	1,177.00***
200	1321	10954	PRINTER MECHANISM	220.00	
			THERMAL PRINTER	80.00	
			DIGITAL CLOCK	625.00	
			CHARACTER GENERATOR	550.00	
				1,475.00*	1,253.75*
			TOTAL SALESMAN NO 1321	1,475.00**	1,253.75**
	9832	18349	DISPLAY LEDS	155.00	
			VIDEO BOARD	195.00	
				350.00*	332.50*
			TOTAL SALESMAN NO 9832	350.00**	332.50**
			TOTAL BRANCH NO 200	1,825.00***	1,586.25***
			FINAL TOTAL	$3,085.00****	$2,763.25****

(a)

FIG. 3. (a) Correct program output. (b) Program output with high-level bug. (c) Program output with low-level bug.

The high-level bug was introduced into the program by removing line 299, which resets the branch-change flag, and the low-level bug by removing line 308, which resets the customer-change flag, and placing the period at the end of line 307. Figures 3b and c present the outputs from the program with the high bug and the low bug, respectively. Eight programmers (four classified as experts and four as novices) debugged each program version. They were given the program listing, a copy of some input data, and the associated output, both correct and incorrect.

3.2. PROCEDURE

Subjects undertook program debugging, speaking aloud as they did so. Their verbalizations were tape-recorded. Subjects first debugged a practice program so they would

15/10/81 SALES REPORT PAGE 1

BRANCH NO	SALESMAN NO	CUSTOMER NO	PRODUCT DESCRIPTION	SALES AMOUNT	DISCOUNTED AMOUNT
100	1225	32911	AUDIO INTERFACE	500.00	
			KEYBOARD	100.00	
			POWER SUPPLY	50.00	
				650.00*	585.00*
		40015	CRT INTERFACE	75.00	
			FLOPPY CONTROLLER	125.00	
			POWER TRANSFORMER	50.00	
				250.00*	250.00*
			TOTAL SALESMAN NO 1225	900.00**	835.00**
	4199	24151	4K RAM	330.00	
			ROM MEMORY	30.00	
				360.00*	342.00*
			TOTAL SALESMAN NO 4199	360.00**	342.00**
			TOTAL BRANCH NO 100	1,260.00***	1,177.00***
200	1321	10954	PRINTER MECHANISM	220.00	
200	1321	10954	THERMAL PRINTER	80.00	
200	1321	10954	DIGITAL CLOCK	625.00	
200	1321	10954	CHARACTER GENERATOR	550.00	
				1,475.00*	1,253.75*
			TOTAL SALESMAN NO 1321	1,475.00**	1,253.75**
200	9832	18349	DISPLAY LEDS	155.00	
200	9832	18349	VIDEO BOARD	195.00	
				350.00*	332.50*
			TOTAL SALESMAN NO 9832	350.00**	332.50**
			TOTAL BRANCH NO 200	1,825.00***	1,586.25***
			FINAL TOTAL	$3,085.00****	$2,763.25****

(b)

FIG. 3. Continued.

be familiar both with the procedure and with verbalizing while debugging. The protocol data was transcribed by a secretary from tape to paper in the form of a series of short, numbered phrases. According to Newell and Simon (1972, p. 166), each phrase should correspond to a naive assessment of what constitutes a single task assertion or reference by the subject. Breaking protocols into small phrases allows a series of relatively unambiguous "measurements" of what information the subject had at a particular time.

3.3. VERBAL PROTOCOL ENCODING

The basis for examining problem-solving processes is the episode: a group of task assertions related to the same goal or objective (Newell & Simon, 1972, p. 84). A subject's protocol consists of a sequence of such episodes, each associated with the

15/10/81 SALES REPORT PAGE 1

BRANCH NO	SALESMAN NO	CUSTOMER NO	PRODUCT DESCRIPTION	SALES AMOUNT	DISCOUNTED AMOUNT
100	1225	32911	AUDIO INTERFACE	500.00	
			KEYBOARD	100.00	
			POWER SUPPLY	50.00	
				650.00*	585.00*
		40015	CRT INTERFACE	75.00	
		40015	FLOPPY CONTROLLER	125.00	
		40015	POWER TRANSFORMER	50.00	
				250.00*	250.00*
			TOTAL SALESMAN NO 1225	900.00**	835.00**
	4199	24151	4K RAM	330.00	
		24151	ROM MEMORY	30.00	
				360.00*	342.00*
			TOTAL SALESMAN NO 4199	360.00**	342.00**
			TOTAL BRANCH NO 100	1,260.00***	1,177.00***
200	1321	10954	PRINTER MECHANISM	220.00	
		10954	THERMAL PRINTER	80.00	
		10954	DIGITAL CLOCK	625.00	
		10954	CHARACTER GENERATOR	550.00	
				1,475.00*	1,253.75*
			TOTAL SALESMAN NO 1321	1,475.00**	1,253.75**
	9832	18349	DISPLAY LEDS	155.00	
		18349	VIDEO BOARD	195.00	
				350.00*	332.50*
			TOTAL SALESMAN NO 9832	350.00**	332.50**
			TOTAL BRANCH NO 200	1,825.00***	1,586.25***
			FINAL TOTAL	$3,085.00****	$2,763.25****

(c)

FIG. 3. Continued.

fulfillment of a specific goal. Hence, the representation of a subject's protocol in episode form captures the goal-oriented behaviour of the subject and the sequence in which it occurs. It can be used, therefore, as the backbone for the representation of the problem-solving process. The *episode outline* is the technique used to define the episode sequence of a protocol. From the episode outline a *strategy diagram* can be derived. This is a higher level abstraction and conceptualization designed to reflect the strategies that programmers use in debugging. The strategy diagram is again abstracted to formulate a *debugging process model.* These three techniques are used here to represent the debugging process. The recording of the debugging processes of subject NH1 is used for illustration purposes in this paper. Subjects are identified by codes. The first character identifies the subject as either an expert or a novice according

to the *ex post* classification. The second character identifies the program bug as either a high-level or a low-level bug. Subjects are further identified, within thse classes, with a numeric character. This subject debugged the program with the high-level bug. The complete set of subject process descriptions, including the three figures and a verbal description for each subject, appears in Vessey (1984, Chapter 7 and Appendix E).

3.3.1. *Episode outline*

Figure 4 presents subject NH1's episode outline. Episodes are determined by the relevance of a given task assertion to the goal in question. New episodes are identified, therefore, by explicit statement of a goal, implicitly by a stated desire to find or to get

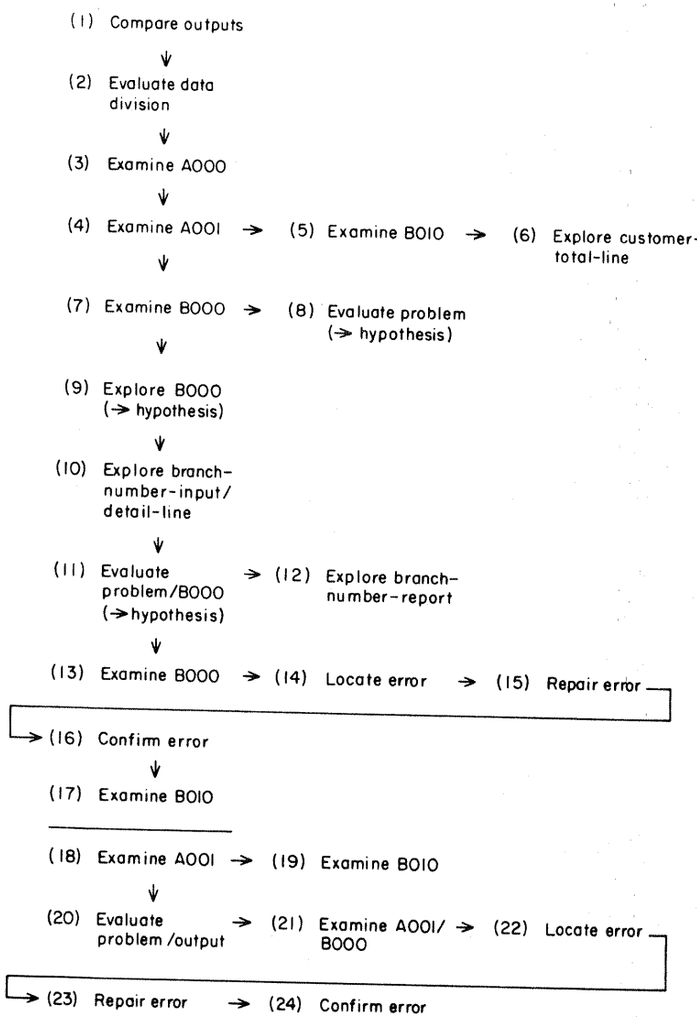

FIG. 4. Episode outline of subject NH1.

a certain item or piece of information, or by a subject focusing attention on another part of the program (e.g. see Newell & Simon, 1972, pp. 283-287). There are two types of relationships between episodes. Dependency-directed relationships, where the second episode occurs as a direct result of the first, are shown diagrammatically via horizontal connections between episodes. For example, there is a dependency-directed association between episodes 4 and 5, and episodes 5 and 6 in Fig. 4 (Shrobe, 1979). Chronological relationships are denoted by vertical connections. This indicates that one episode follows the other in time, but does not occur as a direct result of the first episode. Dependency often can be identified when the subject refers to the same data item or feature of the PROCEDURE DIVISION in consecutive episodes.

Most episodes follow each other in time without being otherwise related. Dependency relationships usually occur when the subject checks on a data item in the WORKING-STORAGE SECTION that has aroused curiosity while examining the PROCEDURE DIVISION. Often the sequence of events preceding finding, correcting, and confirming the error is also dependent in nature (see episodes 13-16 and 20-24 in Fig. 4). Dependency also arises when the subject's evaluation of the situation results in the

A. Determine problem

→ compare correct and incorrect outputs
- repeated applications of:
  'get next item from incorrect output'
  'compare with corresponding item from correct output'
- if not the same, then
  'list differences'

B. Gain familiarity with program

→ scan program listing
- repeated applications of:
  'examine next program section (module)'
  'examine specific module'
  'explore specific W-S item'
  'evaluate problem (→ hypothesis)'

C. Repair error

Locate error
Repair error
Confirm error
Examine specific module

D. Gain familiarity with program

→ scan procedure division
- repeated applications of:
  'examine specific module'
  'evaluate the problem'

E. Repair error

Locate error
Repair error
Confirm error

FIG. 5. Strategy diagram of subject NH1.

statement of a hypothesis. The hypothesis usually does not direct further investigation nor does it appear to be used in the following episode. This situation is denoted by a vertical connection from the episode prior to the evaluation. Episodes 7, 8 and 9 in Fig. 4 illustrate this situation. Breaks in subjects' episode outlines, represented by short horizontal lines, indicate subjects made incorrect repairs that they presented to the researcher as the solution to the problem. Subject NH1 presented an incorrect repair following episode 17.

3.3.2. Strategy diagram

Figure 5 shows subject NH1's strategy diagram. It shows five major problem-solving phases and is derived from the episode outline by identifying groups of consecutive episodes having a similar overall or strategic goal. For example, a number of episodes may be concerned with examining the functions of a number of modules; the associated strategic goal may be to determine the function or the structure of the program. The strategy diagram, then, illustrates subjects' problem-solving approaches in terms of the sequence of strategic goals they set themselves.

The strategy diagram also defines the hierarchy of sub-goals implicit in the fulfillment of each strategic goal. Strategic goals are put into operation by means of tactical goals that specify how a strategic goal is to be fulfilled. At the lowest level of detail, tactical goals translate into operational goals, which are those identified in episodes. Table 1 shows the types of goals programmers set themselves in debugging. There are four major or strategic goals: (1) to determine the problem with the program; (2) to gain familiarity with the function and structure of the program; (3) to explore program

TABLE 1
Hierarchy of subject goals

Strategic goal	Tactical goal	Operational goal
Determine problem	Compare correct and incorrect outputs	Get next item from incorrect output
		Compare with corresponding item from correct output
		List differences
Gain familiarity	Examine program listing	Examine initial comments
	Examine program control	Examine next program section (module)
		Examine specific program section (module)
		Evaluate problem
Explore program structure and function (program control)	Explore procedure division processing	Explore specific module
		Explore specific working-storage item
	Mentally process data through program	Explore control structure
		Process next module in execution sequence
		Evaluate problem
Repair error	Locate error	Locate code in error
	Repair error	Amend code in error
	Confirm error	

execution and/or program control; and (4) to repair (and confirm) the error. Strategic goals 1 and 4 appear in all subjects' protocols. Goals 2 and 3 both occur frequently in the protocols, although some protocols are best characterized by either gaining familiarity with the program or exploring the program alone. The sequence in which subjects set goals 1 and either 2 or 3 differ. Except when subjects make errors, goal 4 is, of necessity, the last in the problem-solving sequence. Subjects in this study used similar tactical and operational goals when pursuing a given strategy, the only difference being one of degree when subjects followed a more or a less active approach to gaining familiarity with the program and exploring the program.†

3.3.3. Debugging process model

Figure 6 presents NH1's debugging process model. It is a still more generalized representation of a subject's approach to problem-solving. Unlike the episode outline, it is no longer strictly sequential. Instead, it shows the flow of problem-solving at a higher level. It employs the same four major elements, phases or building blocks used in the strategy diagram, together with a fifth, evaluate problem. The evaluate problem phase is used to signal the statement of a hypothesis about the error. Subjects sometimes engage in evaluation which does not lead, however, to the statement of a hypothesis. This situation usually arises as a result of an exploration phase and is, therefore, difficult to distinguish from it; it arises less frequently from gaining familiarity with the program. Hence, exploration also includes evaluation not leading to the explicit statement of a hypothesis. It is apparent that, although evaluation phases are added explicitly to the model, the model is a further generalization from the strategy diagram of the subject's approach to problem solving. It is a pictorial representation showing at a glance similarities and differences in the methods used.

3.4. SUBJECTS

The subjects who participated in this research were practising programmers from the State Government Computer Centre, Brisbane, Queensland. With one exception all the programmers had spent their entire programming careers at the State Government Computer Centre. One person had spent 2 years at another government institution and, at the time of the study, had been employed by the Centre for 15 months. Thus the subjects had homogeneous backgrounds.

3.5. ASSESSING DEBUGGING EXPERTISE

This study used two methods to assess programmer expertise, an *ex ante* method and an exploratory *ex post* method. This approach permitted comparison of the effectiveness of the two methods in distinguishing the more from the less skilled programmers.

3.5.1. An ex ante programmer classification

Manager assessment was the initial (or *ex ante*) method used to obtain a set of eight experts and eight novices for the study (Reilly *et al.*, 1975). This method was chosen primarily on the basis of face validity and convenience. Managers at the State Government Computer Centre (the person first contacted and subsequently others at slightly lower managerial levels) assessed programmers who agreed to be subjects. After an initial discussion of what constituted expertise, it became apparent that the manager's main criterion was the length of time the person had held a programming position; that is, experience rather than ability.

† A study by Gould (1975) suggests, however, that this may not always be the case. Gould reports that his subjects used one of two tactics to determine the problem with the program: (1) they examined the output for clues to the problem (the tactic used by all subjects in the current study); (2) they examined the source listing directly.

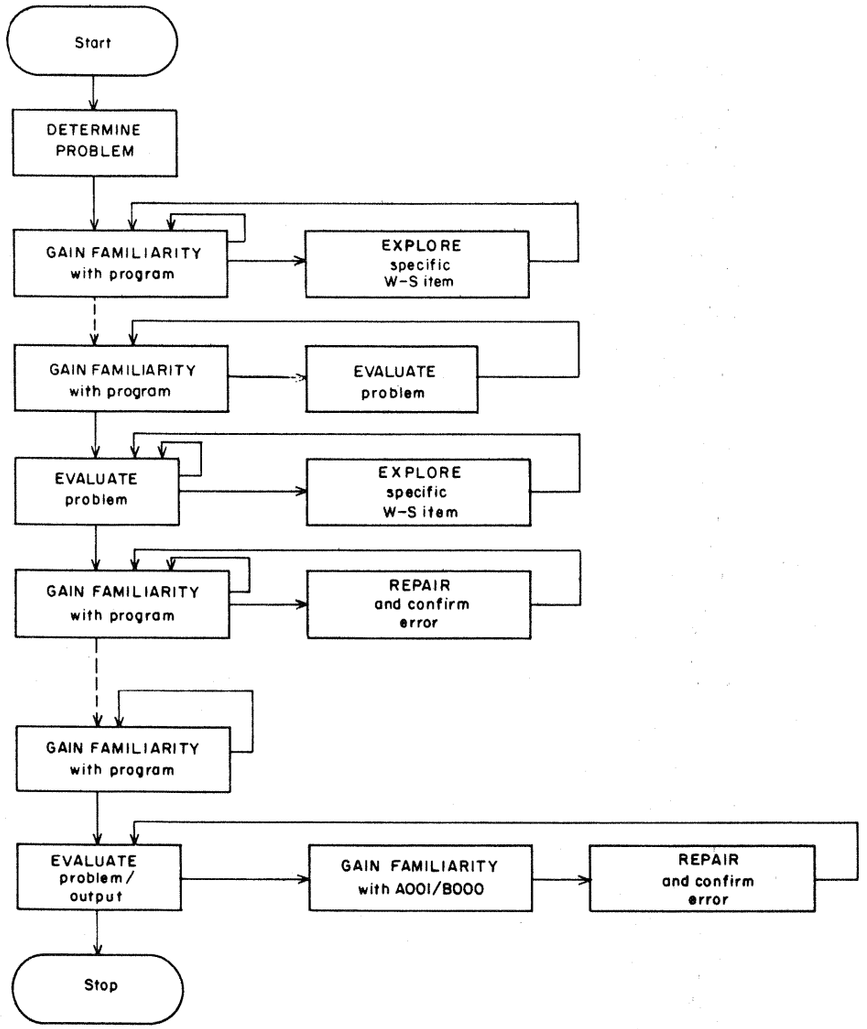

FIG. 6. Model of debugging process of subject NH1.

3.5.2. An ex post programmer classification

The *ex post* classification method used in this study was derived from the debugging efficiency criteria presented in section 2.1. Eight programmers were categorized as

experts and eight as novices according to these criteria, based on a ranking procedure. Since the level of the program bug influenced the number of program position changes programmers exhibited, four programmers who debugged the program with the high bug and four who debugged the program with the low bug were classified as experts in this study; the others were classified as novices.

Table 2 presents the subject classifications based on these three variables as well as the resultant overall designation of the programmer as either an expert or a novice.

TABLE 2
Subject classifications on three debugging performance variables and overall designation

Subject	Function changes	Program debug reversals	Position changes	Classification
EH1	E	E	E	Expert
EH2	E	E	E	Expert
EH3	N	E	E	Expert
EH4	E	E	E	Expert
NH1	N	N	N	Novice
NH2	N	N	N	Novice
NH3	—	N	N	Novice
NH4	N	N	N	Novice
EL1	E	E	E	Expert
EL2	—	E	N	(Expert)
EL3	E	E	E	Expert
EL4	E	E	E	Expert
NL1	N	N	N	Novice
NL2	E	N	N	Novice
NL3	N	N	N	Novice
NL3	N	N	E	Novice

The final classification was derived by assigning subjects to the most frequent class. The three variables classified subjects as experts and novices remarkably consistently. In 11 of the 16 cases, all three variables produced the same classification, while three subjects were rated as either expert or novice on a 2:1 basis. Of the two subjects whose problem solving demonstrated an equal number of function changes across the expert-novice boundary, one was rated twice as a novice on the other variables and so was designated a novice. The other subject, EL2, presented a problem in classification. Since a ranking procedure was used throughout and EL2 was borderline, he was classified as an expert to maintain the balance of eight subjects classed as experts and eight as novices. (This classification also maintained equal numbers on bug type.)

4. Data analysis

Table 3 shows basic subject and task information: the length of work experience, the expert novice classifications and the level of the bug the subject was required to detect and correct, the time taken, the number of words uttered during the experiment, and the verbalization rate in words per second. Note that the subject who accomplished the task in the shortest time and spoke at the fastest rate had only 2 weeks' experience as a practising programmer.

TABLE 3
Basic subject information

Subject	Experience (months)	*Ex ante* Classification	*Ex post* Classification	Bug level	Time Min:S	Time words	Rate (Words/S)
EH1	22·0	Expert	Expert	High	11:00	891	1·35
EH2	12·0	Novice	Expert	High	17:47	837	0·78
EH3	11·0	Expert	Expert	High	14:43	1209	1·37
EH4	2·0	Novice	Expert	High	15:40	1230	1·31
NH1	2·5	Novice	Novice	High	20:50	2170	1·73
NH2	10·0	Expert	Novice	High	19:33	1447	1·23
NH3	24·0	Expert	Novice	High	21:40	1458	1·12
NH4	0·5	Novice	Novice	High	17:20	1107	1·06
EL1	24·0	Expert	Expert	Low	19:23	1259	1·08
EL2	24·0	Expert	Expert	Low	25:29	1910	1·25
EL3	0·5	Novice	Expert	Low	8:40	1047	2·01
EL4	40·0	Expert	Expert	Low	12:40	956	1·26
NL1	33·0	Expert	Novice	Low	38:44	2583	1·11
NL2	10·0	Novice	Novice	Low	31:38	2139	1·13
NL3	36·0	Novice	Novice	Low	36:46	2854	1·29
NL4	0·5	Novice	Novice	Low	37:54	3568	1·57

Three types of analyses were carried out using the verbal data. Firstly, the two programmer classifications were analyzed according to the debugging effectiveness criteria to determine which method should be used for further investigation of debugging processes. Secondly, using this classification, subjects' debugging processes were examined to determine the effects of programmer skill and level of the program bug. Third, independent of the expert–novice classification, a macro analysis was performed that identified strategic decisions the programmers faced in debugging. Programmers were then characterized according to the strategic decisions they made. The expert and novice programmers determined by the first analysis were then compared with the groups of programmers following certain strategic paths derived from the third analysis. This comparison permitted identification of the debugging strategies used by those programmers classified as experts and those classified as novices in this study.

All quantitative data was analyzed using ANOVA and ANCOVA procedures (Nie, Hull, Jenkins, Steinbrenner & Bent, 1975). In all cases there were two factors, the programmer classification and the level of the program bug. The ANCOVA procedure was used when the dependent variable was time. Here, verbalization rate was predicted to have an effect on the outcome; hence it was used as the covariate in the analyses.

4.1. IDENTIFYING EXPERTS AND NOVICES

Table 3 shows that the two programmer classifications assessed in this study classified only 10 of the 16 subjects in the same way. The performance of the two methods was assessed using the debugging effectiveness criteria (debug time and the number of errors subjects made) presented in section 2.2.

The *ex post* programmer classification, which controlled for the chunking ability of programmers, accounted for 73·7% of the variation in debug time compared with 36·1% for the manager classification. The mean debug times according to the *ex post* classification were 15 min 40 s for experts compared with 28 min 3 s for novices, while the corresponding times for the manager classification were 20 min 24 s for experts and 23 min 19 s for novices. Further, the *ex post* classification classified all (five) programmers who made incorrect changes to the program as novices, while the managers classified four of the five programmers as novices. Hence, the *ex post* classification, based on information derived from the verbal protocols, proved to be a better measure of programmer skill for this task than manager assessment.

These results support the concept on which the *ex post* programmer classification is based, namely, that subjects' problem solving processes result in significant variability in performance that is difficult to capture except by explicit recognition of those processes. Further, this result shows that one of the factors that contributes to the variability in subjects' problem solving processes is the chunking ability of programmers. The *ex post* programmer classification, then, was the method used for the succeeding analysis.

4.2. ANALYSIS OF EXPERT AND NOVICE DEBUGGING PROCESSES

The data analysis is presented in terms of variables relating to the outcome or efficiency of debugging, the methods programmers used, and their task-oriented or solution behaviour. The analysis is both quantitative and qualitative in nature.

4.2.1. *Outcome variables*

Table 4 shows several variables related to the outcome or overall conduct of the problem-solving process. Table 5 presents the results of the statistical analysis (ANOVA or ANCOVA) on those variables that are quantifiable.

TABLE 4
Debugging processes—outcome variables

	Subjects	Total Time*	Time to Error†	No. of major phases	No. of episodes	Average time per episode
High Bug	Experts					
	EH1	11:00	9:32	3	18	0:40
	EH2	17:47	15:15	3	15	1:11
	EH3	14:43	10:30	4	20	0:44
	EH4	15:40	10:11	3	12	1:18
	Novices					
	NH1	20:50	19:18	5	24	0:52
	NH2	19:33	17:39	4	21	0:55
	NH3	21:40	20:25	5	27	0:48
	NH4	17:20	16:19	7	22	0:47
Low Bug	Experts					
	EL1	19:23	18:49	5	20	0:58
	EL2	25:29	16:04	5	30	0:51
	EL3	8:40	6:53	4	17	0:30
	EL4	12:40	12:19	4	9	1:24
	Novices					
	NL1	38:44	13:32	6	33	1:10
	NL2	31:38	30:23	8	26	1:13
	NL3	36:46	35:01	10	31	1:11
	NL4	37:54	37:49	7	31	1:13

* All time measures are presented as minutes and seconds.
† The "time to error" was measured by the formula:

$$\frac{\text{number of phrases to error} \times \text{total time}}{\text{total number of phrases}}$$

Total debug time. Total debug time refers to the time taken both to discover the error and subsequently to confirm it. Both the skill level and the bug level significantly affected debug time ($R^2 = 0{\cdot}737$). Novices took longer to debug programs in general than experts ($P < 0{\cdot}001$) and programmers took longer to correct programs with low bugs than with high bugs ($P = 0{\cdot}001$). In addition, there were two interaction effects. As expected, novices took longer to debug the program with the low bug than the high bug and novices took longer than experts for the low bug. This result suggests that the programmer classification method based on subjects' chunking ability, together with bug level, is effective in distinguishing the more able from the less able programmers.

TABLE 5
Statistical results derived from selected outcome variables

Dependent variable	EN effects	Bug effects	Interaction effects	R^2
Total time	0·000 N > E	0·001 L > H	0·009 N > E for L L > H for N	0·737
Time to error	0·005 N > E			0·572
No. of major phases	0·001 N > E	0·006 L > H		0·712
No. of episodes	0·003 N > E			0·570
Average time per episode				0·125

Time to discover the error. This variable refers to the length of time subjects took to articulate the error, but does not include the time to confirm the error. The variable was significant only for the expert-novice classification ($P = 0{\cdot}005$, $R^2 = 0{\cdot}572$). Novices take longer both to discover the error and to discover and confirm the error. This result suggests there may be little difference between programmers in the time to confirm errors.

However, the result for bug level is different from that for total debug time, i.e. time to discover the error is not significantly higher for the low-level bug, as would be expected. The significant result for total debug time probably arises because of the time subject NL1 (with the low bug) required to confirm the error. He found the error in 13 min 32 s but then took almost twice that period to reassure himself that he was correct (25 min 12 s). This result indicates that subject NL1 had not created an adequate model of the program's function and structure prior to indicating the error; he simply did not know how the program worked and could not confirm the error at that time in terms of his internal model of the program. (This aspect is considered further under Outcome variables: system thinking.)

Number of major phases. The number of major problem-solving phases, obtained from the subject's strategy diagrams, varied with both the *ex post* skill classification and the bug level ($R^2 = 0{\cdot}712$). Novices engaged in more major phases in debugging than experts ($P = 0{\cdot}001$), and subjects as a whole engaged in more major phases for low than for high bugs ($P = 0{\cdot}006$). This result is consistent with the number of errors that subjects made in debugging the programs. When making a correction, they entered a repair phase and when told they were not correct, they again resumed their analysis of program structure. In this way, they entered into at least one and probably two more major problem-solving phases. Since the *ex post* classification classified all programmers who made errors as novices, it follows that novices engage in more gross phases than experts during debugging.

Number of episodes. Novices required more episodes than experts to solve the problem ($P = 0{\cdot}003$, $R^2 = 0{\cdot}570$). However, the level of the program bug had no effect on the number of episodes. The result for the skill classification relates both to total debug time and to the average episode time. Since novices took longer in general to debug than experts and since the average episode length did not vary (see next subheading), it follows that novices engaged in more problem-solving episodes than experts.

Average time per episode. Neither the programmer classification nor the bug level significantly affected the average time expended per episode. Experts and novices spent similar amounts of time in examining individual aspects of the problem, and programmers in general engaged in problem-solving episodes of similar length, irrespective of whether they were debugging programs with high or low bugs.

4.2.2. *Method variables*

Table 6 shows variables relating to the method or process subjects used in debugging. Table 7 presents the results of the statistical analysis performed on quantitative variables. One of the most significant outcomes of the process analysis is the realization that all subjects' debugging processes can be described in terms of five major problem solving phases: problem determination, gaining familiarity with the program, exploration of particular aspects, evaluation leading to the statement of a hypothesis and, finally, error repair. The debugging process model, the third technique for recording processes, reflects the type and sequence of phases in which individual subjects engaged. Every protocol does not necessarily display all phases, and certain phases may occur several times during problem-solving. All protocols include, however, both problem determination and error repair phases.

Module examination procedure. Subjects approached the essential task of ascertaining the program structure principally in one of two ways. In the first approach they read through at least the first three modules, A000-CREATE-SALES-REPORT, A001-PROCESS-AND-READ, and B000-PROCESS-DETAIL-RECORDS, in sequence as they appeared in the listing (seven subjects). The second approach was to examine the modules in execution sequence, i.e. A000-CREATE-SALES-REPORT, followed by A001-PROCESS-AND-READ, and then by B010-PROCESS-CUSTOMER-CHANGE (six subjects). Two subjects, EH4 and NH3, engaged in the most active search process and started their investigation of the program structure by looking for the module where they believed the error to be: B000-PROCESS-DETAIL-RECORDS. From then on, however, their approaches differed. Subject EH4 found the error by reference to module B000-PROCESS-DETAIL-RECORDS alone; he then worked backwards through the program listing, referencing first module A001-PROCESS-AND-READ and then A000-CREATE-SALES-REPORT, in order to confirm it. NH3, on the other hand, first followed an execution sequence by glancing briefly at module C000-PRINT-HEADINGS; the third module he referenced was A000-CREATE-SALES-REPORT. The remaining subject, NH4, did not follow a pattern for module examination. He looked first at A000-CREATE-SALES-REPORT, reading out the PERFORM statements for modules A001-PROCESS-AND-READ, B010-PROCESS-CUSTOMER-CHANGE, B020-PROCESS-SALESMAN-CHANGE, and B030-PROCESS-BRANCH-CHANGE (activated when the main body of processing has concluded), interspersed with two references to the WORKING-STORAGE SECTION. Next he examined B000-PROCESS-DETAIL-RECORDS.

Since the module examination procedure investigates only the sequence in which subjects approach the early stages of the task, few differences would be expected for

TABLE 6

Debugging processes—method variables

	Subject	Module examination procedure*	Familiarity before problem determination†	Familiarity and exploration before error	Number of different modules examined	Number of times B000 examined	Number of DATA DIVISION items examined
High Bug	Experts						
	EH1	2		E	6	2	0
	EH2	1		F	4	3	2
	EH3	1		F	6	4	4
	EH4	3		E	3	3	2
	Novices						
	NH1	2		F	4	4	3
	NH2	1		F and E	6	7	1
	NH3	3		E	7	5	4
	NH4	4		F	6	4	3
Low Bug	Experts						
	EL1	1	15·56	F and E	5	5	2
	EL2	1	32·93	F and E	4	6	4
	EL3	1	9·80	F and E	4	4	1
	EL4	2		F and E	4	2	1
	Novices						
	NL1	1		F and E	7	10	3
	NL2	2		F and E	6	8	1
	NL3	2		F	5	8	5
	NL4	2		F	5	5	3

* Subjects approached the debugging task essentially in four ways, determined principally from the first three modules they examined:
1 = lexical sequence; A000-CREATE-SALES-REPORT, A001-PROCESS-AND-READ, B000-PROCESS-DETAIL-RECORDS
2 = execution sequence; A000-CREATE-SALES-REPORT, A001-PROCESS-AND-READ, B010-PROCESS-CUSTOMER-CHANGE
3 = task solution, first module = B000-PROCESS-DETAIL-RECORDS
4 = indeterminate sequence; A000-CREATE-SALES-REPORT, B010-PROCESS-CUSTOMER-CHANGE, B000-PROCESS-DETAIL-RECORDS

TABLE 7

Statistical results derived from selected method variables

Dependent variable	EN effects	Bug effects	R^2
No. of different modules examined	0·045 N > E		0·299
No. of times B000 examined	0·007 N > E	0·023 L > H	0·578
No. of DD items examined			0·104

bug level. There are differences, however, in the methods used by experts *vis-à-vis* novices. Table 8 summarizes the results. Experts, in general, are more relaxed about debugging (situation-dependent-problem solving). They are content to read through the program as it unfolds. Again, this is an illustration of the high-level problem-solving that so often appears to characterize the behaviour of experts. Novices, on the other hand, prefer to assess how the program executes sooner than experts.

TABLE 8

Summary of module examination procedure

Procedure	Experts	Novices
Lexical	5	2
Execution	2	4
Solution	1	1
Indeterminate		1

Familiarity before problem determination. Three subjects (EL1, EL2, and EL3), all classed as experts in this study, gained some familiarity with the program before comparing the correct and incorrect outputs to discover the problem with the program. Subject EL3 read the introductory comments only (one episode, 9·80% of total phrases). Subject EL1 read the initial comments, reviewed the FILE and WORKING-STORAGE SECTIONs of the DATA DIVISION, and then read the comments relating to the first two modules (A000-CREATE-SALES-REPORT and A001-PROCESS-AND-READ). This initial familiarization involved four episodes and amounted to 15·56% of the total phrases uttered. Subject EL2 engaged in an extended initial familiarization phase that consumed 14 episodes representing 32·93% of the complete problem solving effort. He looked first at the DATA DIVISION, then at modules A000-CREATE-SALES-REPORT, A001-PROCESS-AND-READ, and B000-PROCESS-DETAIL-RECORDS. While perusing the PROCEDURE DIVISION, he frequently referred to items in the WORKING-STORAGE SECTION.

Familiarity and exploration before error. Most subjects (with the exception of those discussed in the previous section) first assessed the problem with the program by examining the correct and incorrect outputs, generally on a line-by-line basis. This was usually followed by a familiarization phase where subjects read through parts of

the program to discover what it was doing. If subjects did not detect the error by simply reading through the program, they usually engaged in active exploration of the program in the form of mental execution. Exploration reveals information on the execution sequence and on the values of data and control variables.

The protocols of NH3 and EH4 contain no familiarization phase, while that of EH1 was very short and is classed as exploration only. Certain subjects found the error without engaging in exploration, i.e. active searching for certain structures in accordance with a hypothesis, implicit or explicit, or mentally executing the program to determine how it was functioning. These include (in the sequence in which they appear in Table 6) EH2, EH3, NH1, NH4, EL3, NL3, and NL4. Of these, subjects NH1, NL3 and NL4 had considerable difficulty in finding the error. The remaining subjects, those who did not find the error after reading through the relevant modules once or twice, turned to exploration, most frequently in the form of mental execution of the program (NH2, EL4, EL1, EL2, NL1, and NL2). They generally concentrated on control aspects such as resetting the previous numbers and the values of the change flags.

TABLE 9
Summary of familiarity and exploration phases before error detection

	Phases	Experts	Novices
High bug	Familiarity	2	2
	Exploration	2	1
	Familiarity and Exploration	0	1
Low bug	Familiarity	1	2
	Exploration	0	0
	Familiarity and Exploration	3	2

Table 9 summarizes the use of familiarity and exploration phases before bug detection for both the skill classification and bug level. No consistent patterns of differences between groups can be identified.

Number of different modules examined. One of the criteria for derivation of the *ex post* programmer classification used in this study was that novices could not chunk programs as efficiently as experts. They would engage, therefore, in more erratic problem-solving behaviour than experts, illustrated by the frequency of their changes of reference positions in the program. Since low bugs incurred more position changes than high bugs, confirming the greater difficulty of locating and correcting the program with the low bug, bug level was controlled in deriving the classification. Inherent in this classification, therefore, is the fact that novices make changes more frequently than experts in the material they reference.

In the process analysis, the variable investigated is the number of different modules that programmers reference in debugging. Only a few modules are relevant to understanding the program structure. Modules A001-PROCESS-AND-READ and B000-PROCESS-DETAIL-RECORDS are those in which the flags are set and unset; in subordinate modules B010-PROCESS-CUSTOMER-CHANGE, B020-PROCESS-SALESMAN-CHANGE, and B030-PROCESS-BRANCH-CHANGE, the "previous"

numbers are reset for matching purposes. These three sets of modules perform all control functions in the program. As long as the problem is characterized as a control problem, these are the modules where the error(s) might be expected to occur. The only other possibility is that DETAIL-LINE is not cleared before printing. As noted previously, however, closer examination would show that the problem is not uniform, i.e. it does not occur all the time, and so cannot be one of clearing DETAIL-LINE. Also, one would logically expect that clearing DETAIL-LINE would be accomplished within module B000, which carries the title B000-PROCESS-DETAIL-RECORDS. Hence, it is not essential to reference modules other than the controlling module A000-CREATE-SALES-REPORT, A001-PROCESS-AND-READ, B000-PROCESS-DETAIL-RECORDS, and the three "change" modules. The number of different modules that programmers reference can therefore be regarded as a measure of the confidence that programmers have in looking at what they consider to be the relevant modules. Hence, it is expected that the less confident programmers (novices) will reference more modules than the more confident programmers (experts). This reasoning is supported by the result: novices examine more modules than experts ($P = 0{\cdot}045$, $R^2 = 0{\cdot}299$). Bug level has no effect on the number of modules that programmers reference.

Number of times B000 was examined. The module in error is B000-PROCESS-DETAIL-RECORDS. Novices reference module B000-PROCESS-DETAIL-RECORDS significantly more often than experts ($P = 0{\cdot}007$) and programmers reference B000-PROCESS-DETAIL-RECORDS more often for low-level bugs than for high-level bugs ($P = 0{\cdot}023$, $R^2 = 0{\cdot}578$). These results are similar to those for the number of different modules that programmers examine. They demonstrate lesser ability to grasp the control relationships established in the program and the interrelationships between modules.

Number of DATA DIVISION items examined. Again, because they are less confident than experts, it might be expected that novices would refer to items in the DATA DIVISION more frequently. However, no such differences were observed. Similarly bug level was not significant ($R^2 = 0{\cdot}104$).

4.2.3. Solution variables

Table 10 shows variables directly related to the solution process. Table 11 presents the results of the statistical analysis on readily quantifiable variables.

Number of hypotheses. Nine subjects stated hypotheses ranging from one to three in number. Novices stated more hypotheses than experts ($P = 0{\cdot}045$; $R^2 = 0{\cdot}230$). Perhaps experts have automated their problem-solving processes to a greater extent than novices and hence do not state hypotheses as frequently during debugging. Alternatively, since novices make more errors (see later), they will consider more possible causes of the problem.

Types of hypotheses. Table 10 presents the hypotheses that programmers articulated. They range from the general "control break problem" to resetting the previous number(s), moving SPACES to DETAIL-LINE, and not setting or resetting a control flag (see Appendix A). Of a total of 19 hypotheses, three related to control break, five to resetting the previous number(s), nine to clearing DETAIL-LINE, and two to resetting the change flag. Activity that resulted from understanding the program structure and

led directly to error correction was considered to be evaluative in nature rather than hypothesis activity. Only one person hypothesized (twice) that the change flag was the problem (subject EH3). It is apparent, therefore, that in debugging stating the correct hypothesis is not a prerequisite to finding the bug. Subjects may have made implicit assumptions about the possible cause of error that may or may not have been correct. However, only one subject made the correct explicit assumption. This subject was classed as an expert. Other studies suggest that experts make good first guesses about the solution to a problem. This research found that experts did not make better first guesses, nor did they make more guesses. The crucial factor in debugging performance is that experts were not as committed to their hypotheses as novices. Therefore, they were not blind to new information.

Problem-solving constraints. Several subjects stated a hypothesis but did not actively evaluate it, preferring to let the problem unfold as they became more familiar with the program. These subjects are designated in Table 10 as "unconstrained", and include EH1, EH3, NH2, and NH4. Others, however, stated a hypothesis early in task execution and were so determined they were correct that they failed either to understand the program structure or to evaluate their proposed change. These include NH1, NL2, NL3, and NL4. They are designated in Table 10 as constrained.† In certain cases they did not recognize signals that their hypotheses may have been incorrect, showing inflexibility in adopting and discarding hypotheses (NH1, NL2, and NL4). Two subjects, NL2 and NL3, used a "shotgun" approach to error detection that was not related to hypothesis generating activity alone. They made continual changes to the program in the hope of eventually producing the correct one; i.e. they considered the onus of decision was on the researcher to accept or reject the changes rather than on themselves to justify their corrections. These subjects were all classified as novices.

System thinking. Experts, whether they stated hypotheses or not, gradually created an implicit model of program structure and function, which permitted them to place the error in context. Those subjects who found the error without creating the model of program structure and function (e.g. subjects NL1, EL3 and EH4) found it essential to create the model before being satisfied they had found the error. This is an example of what Johnson, Hassebrock, Duran & Moller (1982, p. 226) call "system thinking".

Those subjects who are not regarded as perceiving the problem from a system viewpoint are NL1, NH3, and NH4. Although subject NL1 eventually constructed such a model, he took twice as long to construct the model as he did to indicate the error and is therefore considered to be deficient in his ability to think in system terms.

Program structures considered. Subjects explicitly examined a number of program structures in their search for the error. To some extent these structures are reflected in the hypotheses that subjects articulated, but they did not always state specifically their perceptions of the cause of the error. Two cases in point are subjects NH3 and EL1. They made single task assertions, such as "therefore that cannot be the problem", when they found a structure they obviously had thought might have been missing from the program. Such entries are made in brackets. Including these structures, six subjects explicitly considered previous numbers and 12 subjects considered spaces and change

† This type of approach to problem-solving is termed "depth-first"by Nilsson (1980) and "extraction" by Feltovich (1981). It is characterized by rejection of the suspected problem only when necessary. The alternative problem solving approach is "breadth-first" or "precautionary" (Nilsson and Feltovich, respectively).

TABLE 10

Debugging processes—solution variables

	Subjects	Number of hypotheses	Types of hypotheses*	Problem solving constraints†	System thinking	Program-structures considered	Number errors	Types of error
High bug	Experts							
	EH1	3	CB, SPN, S	0		PN, S, F		
	EH2	—			—			
	EH3	3	CB, F, F	0		F		
	EH4	2	S, S			S, F		
	Novices							
	NH1	2	S, S	C		S, F	1	S
	NH2	3	CB, SPN, S	0		PN, S, F		
	NH3	—			No	PN, (S)F	1	SPN
	NH4	1	SPN	0	No	PN, F		
Low bug	Experts							
	EL1	—				(S), (PN), F		
	EL2	—				—		
	EL3	—				PN, F		
	EL4	—				F		
	Novices							
	NL1	3	SPN, S, SPN		No	PN, S, F	2	SPN, SPN
	NL2	—		C, S		PN, F	2	TPN, S
	NL3	1	S	C, S		S, PN	2	S, S
	NL4	1	S	C		S	2	

* The errors made, types of program structures and hypotheses considered, take the following forms:
CB = control break
F = flag
PN = previous number
SPN = set previous number
TPN = test previous number
S = spaces
Entries in brackets for the "program structures considered" column indicate a one line reference to a structure such as "move spaces; so they must be getting moved back in".
† This field refers to the degree to which subjects were committed to their hypotheses.
O = unconstrained
C = constrained
S = shotgun

TABLE 11
Statistical results derived from selected solution variables

Dependent variable*	EN effects	R^2
Number of hypotheses	0·045 N > E	0·230
Number of mistakes	0·005 N > E	0·500

* There were no bug effects for the solution variables.

flags. Note the bias in the number of subjects who considered flags since this was the error; hence, everyone eventually referred to flags as being the source of error. Only two subjects, EH3 and EL4, considered change flags alone, while two more, EH2 and EL2, appeared to detect the bug with no explicit consideration of structures of any kind. Two subjects, who had previously considered other structures, did not finish with an explicit consideration of change flags: NL3 and NL4. Subject NL3 suggested the correct amendment, together with other changes he had not deleted, as yet another amendment that could have made the program work. NL4 appeared just to state the correct solution; he had already committed an error at that point.

Number of mistakes. Programmers classed as novices made significantly more errors than those classed as experts ($P = 0{\cdot}005$, $R^2 = 0{\cdot}500$). Bug level had no effect although six of the eight mistakes were committed for the low-level bug.

Types of mistakes. Subjects made limited sorts of mistakes (as reported in Appendix A). Of eight mistakes, four involved moving SPACES to DETAIL-LINE (or to some part of DETAIL-LINE), and the other four involved branch, salesperson or customer numbers. Three of these latter mistakes involved resetting the previous numbers, while the fourth introduced an unnecessary test to determine whether a number had changed prior to printing that part of the DETAIL-LINE repeatedly written in error.

4.3. ANALYSIS OF SUBJECTS' DEBUGGING STRATEGIES

Figure 7 presents a pictorial representation of the strategy paths the programmers followed. The representation of strategy paths differs from the individual subjects' strategy diagrams in that it describes at a macro level the strategies of all subjects. The strategy paths are characterized by four binary factors leading to a possible 16 paths. These four variables represent significant elements in the subjects' debugging processes. They derive from the previous analysis. The binary variables, in the sequence in which subjects considered them (explicitly or implicitly), are:

(1) Whether subjects examined the program or the output first (Table 6: Familiarity before problem determination).

(2) Whether subjects engaged in active or passive examination of the problem (Table 6: Module examination procedure).

(3) Whether subjects were constrained by the hypotheses they stated (Table 10: Problem-solving constraints).

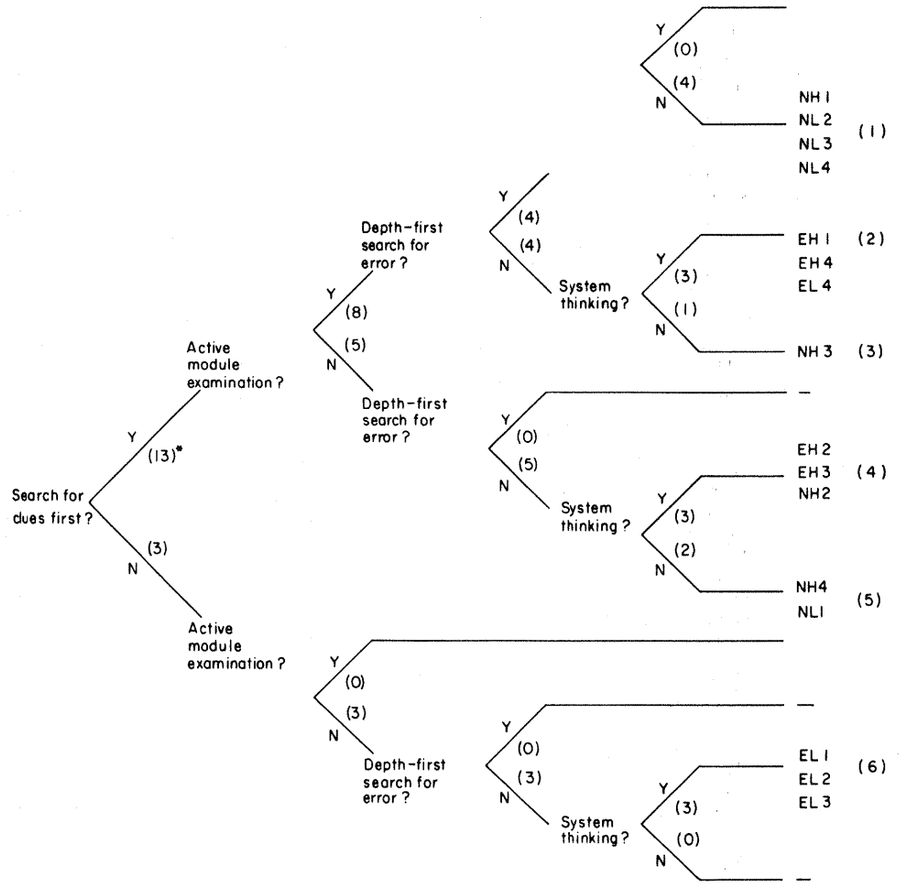

FIG. 7. Strategy paths followed by programming subjects.

* The numbers in brackets on the branches represent the number of subjects following that strategy.

[1] The alternative to searching first for clues to the problem is to examine the program structure and function and then to search for clues.

[2] Active module examination is distinguished by: (a) initially following the execution path of the program rather than the lexical sequence; or (b) actively searching for the error rather than first undestanding the program.

[3] All subjects who were not recorded as being constrained by their hypotheses were regarded as engaging in breadth-first search for the error.

(4) Whether subjects developed a model of the program structure and deduced a causal model of the error (Table 10: System thinking).

The strategies are represented in the form of a decision tree (DeMarco, 1979; Gane & Sarson, 1979), with the intention of representing *temporally* the strategic decisions made by subjects. The numbers of subjects choosing each path is represented on the diagram. Subjects followed six of the 16 paths.†

†The strategies are numbered to the right of Fig. 7.

Examination of the subjects following each strategy shows that strategies 1, 3 and 5 are followed principally by subjects classified as novices according to the expert-novice programmer classification, while strategies 2, 4 and 6 are followed principally by those classified as experts. Reformulation of the decision tree presented in Fig. 7 produces the complete and consistent decision table of Table 12 (Gildersleeve, 1970). It shows that two factors determine expert behaviour in this diagnostic task: the ability to pursue a breadth-first search for the error† and the ability to think in systems terms. Programmers who are constrained by the hypotheses they generate are novices. Further, programmers who engage in breadth-first search for the error but who do not formulate a model of the program structure and conceive of the error within that context will be likely to make mistakes and will therefore be regarded as novices. Whether subjects initially examine the output of the program has no effect on problem-solving. Neither does reading modules vs mentally executing modules.

TABLE 12
Decision table for determining expert and novice subjects according to the ex post programmer classification

		Rules		
		1	2	3
Conditions	(1) Breadth-first search for error	Y	Y	N
	(2) System thinking	Y	N	—
Actions	(a) Designate expert	X		
	(b) Designate novice		X	X

* This table approaches the designation of experts and novices from the viewpoint of experts as opposed to Fig. 7, which approached it from the viewpoint of novices. Figure 7 derived from the analysis in this chapter which identified constrained problem-solving as a characteristic of novices, while a more positive approach identifies the characteristics of experts.

The decision table, based on only two binary conditions, classifies 15 of the 16 programmer subjects in the same manner as the skill classification, which is based on the chunking ability of the subjects. The sixteenth subject is NH2. Perusal of NH2's process description (Vessey, 1984, Appendix E.2) shows that there is little in his protocol to suggest that he is a novice according to the criteria presented in this section. He does not exhibit, however, a very refined chunking ability (see Table 2). He is ranked tenth in a three-way tie on function changes, ninth in a three-way tie on Program Debug reversals, and eleventh in a two-way tie on position changes.

5. Implications of the results

The objective of this research was to determine those characteristics of programmers' debugging processes that lead to debugging expertise.

† For further discussion of the significance of breadth-first vs depth-first approaches to both diagnostic and design problems, see Feltovich (1981), Jeffries *et al.* (1980), Johnson, Duran, Hussebrock *et al.* (1981), Malhotra *et al.* (1980), and Nilsson (1980).

5.1. IMPLICATIONS FOR DEBUGGING PROCESSES

Tables 13 and 14 present summaries of the differences in debugging processes assessed quantitatively for level of program bug and the exploratory *ex post* programmer classification, respectively. Differences in debugging processes were observed between bug levels when subjects made mistakes. Mistakes led to increases in the number of phases in which programmers engaged. Mistakes were generally associated with the more deeply entrenched low-level bug. Programmers did not otherwise appear to modify their problem-solving methods for the low bug. There were again differences in the effectiveness of the application of those methods as a result of the differing bug complexity. This is evidenced particularly in the time required to debug the two programs.

TABLE 13
Summary of the effects of bug level on process variables

Dependent variable	Direction
Debug time	L > H L > H for N
Position changes	L > H
Major phases	L > H
B000 examinations	L > H

TABLE 14
Summary of the effects of skill level on process variables

Dependent variable	Direction
Debug time	N > E N > E for L
Time to error	N > E
Major phases	N > E
Episodes	N > E
High-level module examination	E > N
Familiarity before problem determination	E > N
Modules examined	N > E
B000 examinations	N > E
Mistakes	N > E

All programmers engaged in similar types of activity during debugging, i.e. all programmers' debugging processes could be described with five basic building blocks. There are certain differences in the way the activities are sequenced and whether or not a subject employs a given activity. The overriding consistent difference in expert–novice processes that emerges from this study is the preference of expert programmers to work at a high level without apparent concern for solving the problem. Novices are anxious about their ability to solve the problem. They tend to focus directly on getting a solution rather than understanding the program and how it functions. They are inflexible in their approach to the problem and their (proposed) solution to it. From the subjects' strategy diagrams, it appears that novices have the same basic methods

available to them but that there are differences in the effectiveness of the application of these methods.

5.2. IMPLICATIONS FOR A CONCEPT OF PROGRAMMING EXPERTISE

The *ex post* programmer classification, based on subjects' ability to chunk programs, together with bug level, explained 73·7% of the variation in debug time and classified all programmers who made mistakes as novices.

Further support for the use of chunking ability as a measure of debugging expertise was provided by the analysis of subjects' strategy paths. Except for subject NH2, classification of subjects according to their high-level problem solving capabilities and their approach to modelling the system resulted in the same programmer classification as that based on chunking ability. Hence, a micro-analysis of debugging activities and a macro-analysis of debugging strategies essentially produced similar results. Two diverse methods resulting in convergent programmer classifications lend support to the notions that underlie those methods and hence provide insight into the nature of debugging expertise.

Expert debuggers are those who can chunk programs more effectively. They exhibit disciplined approaches to problem-solving, pursuing similar types of behaviour rather than frequently changing mode of behaviour, checking on the clues to the problem and changing reference points within the program. Furthermore, expert debuggers are those who approach the problem in a relaxed manner. They do not permit the formulation of hypotheses to lead them to a depth-first search for the error. Instead, they allow the structure of the program to unfold, place the clues in the context of that structure, and conceptualize the error in terms of the program structure. Directed search for the solution to the problem in terms of initial examination of the output for clues to the problem and/or the module in error is not a determinant of debugging expertise.

The type of problem-solving outlined above—i.e. breadth-first, keeping constraints open—is behaviour commonly found to characterize the problem-solving of experts. In addition, it is behaviour that Dreyfus (1982) refers to as situation-dependent behaviour. Problem-solvers who are constrained by their initial hypotheses do not always react to the program content but perceive what they expect to perceive. They are therefore situation-independent. So too are those programmers who do not develop a causal model of the program structure and the error in it, i.e. those who do not exhibit "system thinking". This study provides no support, however, for the notion of a formal symptom-pattern recognition feature such as that found in medical diagnosis (e.g. see Bouwman, 1978).

6. Limitations of the research

The major limitation of the study is that the reliability of the method used to classify programmers has not been tested independent of the current data. The study shows that, in a given set of circumstances, one of the primary factors associated with variable programming performance is the chunking ability of programmers. The *ex post* classification method should now be tested to establish whether it classifies subjects consistently in the same manner. That is, a test-retest examination of the method is required to assess the reliability.

7. Conclusions

This research provides insights into the nature of debugging expertise and hence contributes to a general theory of programming expertise. General empirical propositions about the expertise required to repair programs should be formulated from the theory and the strategic propositions tested.† This research suggests that some of the strategic propositions to be tested in the investigation of debugging expertise are:

1. (a) Experts use breadth-first approaches to problem-solving and, at the same time, adopt a system view of the problem area;

(b) Experts are proficient at chunking programs and hence display smooth-flowing approaches to problem-solving.

2. (a) Novices use breath-first approaches to problem-solving but are deficient in their ability to think in system terms;

(b) Novices use depth-first approaches to problem-solving;

(c) Novices are less proficient at chunking programs and hence display erratic approaches to problem-solving.

Further investigation will serve to extend and refine the theory and also to set boundaries on the applicability of the strategic propositions.

The author is indebted to Gordon Davis, Vasant Dhar, Ron Weber, and participants in workshops at the University of Minnesota and New York University for comments on earlier versions of this paper.

References

ANDERSON, J. R., GREENO, J. G., KLINE, P. J. & NEVES, D. M. (1981). Acquisition of problem-solving skill. In Anderson, J. R., Ed., *Cognitive Skills and Their Acquisition.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

ATWOOD, M. E. & RAMSEY, H. R. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging. *NTIS*, AD-A060 522/0.

BIGGS, S. F. (1978a). An investigation of the decision processes underlying the assessment of corporate earning power. *Unpublished Doctoral Dissertation*, University of Minnesota.

BIGGS, S. F. (1978b). An empirical investigation of the information processes underlying four models of choice behavior. In Burns, T. J., Ed., *Behavioral Experiments in accounting. II.* College of Administrative Science, The Ohio State University.

BOUWMAN, M. J. (1978). Financial diagnosis: A cognitive model of the processes involved. *Unpublished Doctoral Dissertation*, Carnegie-Mellon University.

BOUWMAN, M. J. (1983). Human diagnostic reasoning by computer: An illustration from financial analysis. *Management Science*, **29**, 653–672.

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

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

CHI, M. T. H., FELTOVICH, P. J., & GLASER, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, **5**, 121–152.

DEMARCO, T. (1979). *Structured Analysis and System Specification.* Englewood Cliffs, New Jersey: Prentice-Hall.

† According to Dubin (1978, p. 168), "Strategic propositions are those that state critical or limiting values for one of the units involved", and further (p. 210), "If strategic propositions (do) not produce positive results, then there (is) no point in worrying about the other testable propositions". (The verbs in brackets have been changed from past to present tense.)

DREYFUS, S. E. (1982). Formal models vs human situational understanding: Inherent limitations on the modeling of business expertise. *Office: Technology and People*, **1**, 133–165.
DUBIN, R. (1978). *Theory Building*. Revised edn. New York: The Free Press.
FELTOVICH, P. J. (1981). Knowledge based components of expertise in medical diagnosis. *Unpublished Doctoral Dissertation*, University of Minnesota.
GANE, C. & SARSON, T. (1979). *Structured Systems Analysis*. Englewood Cliffs, New Jersey: Prentice-Hall.
GILDERSLEEVE, T. R. (1970). *Decision Tables and Their Practical Application in Data Processing*. Englewood Cliffs, New Jersey: Prentice-Hall.
GOULD, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, **7**, 157–182.
GOULD, J. D. & DRONGOWSKI, P. (1974). An exploratory study of computer program debugging. *Human Factors*, **16**, 258–277.
JEFFRIES, R., TURNER, A. A., POLSON, P. G. & ATWOOD, M. E. (1981). The processes involved in designing software. In Anderson, J. R., Ed., *Cognitive Skills and Their Acquisition*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
JOHNSON, P., DURAN, A., HASSEBROCK, F., MOLLER, J., PRIETULA, M., FELTOVICH, P. & SWANSON, D. (1981). Expertise and error in diagnostic reasoning. *Cognitive Science*, **5**, 235–283.
JOHNSON, P., HASSEBROCK, F., DURAN, A. & MOLLER, J. (1982). Multimethod study of clinical judgment. *Organizational Behavior and Human Performance*, **30**, 201–230.
KINTSCH, W. and VAN DIJK, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, **85**, 363–394.
LARKIN, J. H. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. In Anderson, J. R., Ed., *Cognitive Skills and Their Acquisition*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
LARKIN, J. H., MCDERMOTT, D., SIMON, D. P. & SIMON, H. A. (1980). Expert and novice performance in solving physics problems. *Science*, **208**, 1335–1342.
LEWIS, C. H. (1981). Skill in algebra. In Anderson, J. R., Ed., *Cognitive Skills and Their Acquisition*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
MALHOTRA, A., THOMAS, J. C., CARROLL, J. M. & MILLER, L. A. (1980). Cognitive processes in design. *International Journal of Man-Machine Studies*, **12**, 119–140.
MAYER, D. B. & STALNAKER, A. W. (1968). Selection and evaluation of computer personnel—The research history of SIG/CPR. *Proceedings of the 23rd ACM National Conference*, 657–670.
MYERS, G. J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, **21**, 760–768.
NEWELL, A. & SIMON, H. A. (1972). *Human Problem Solving*. New York: Prentice-Hall.
NIE, N. H., HULL, C. H., JENKINS, J. G., STEINBRENNER, K. & BENT, D. H. (1975). *Statistical Package for the Social Sciences*. 2nd edn. New York: McGraw Hill.
NILSSON, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, California: Tioga.
PAYNE, J. W., BRAUNSTEIN, M. L. & CARROLL, J. S. (1978). Exploring predecisional behavior: An alternative approach to decision research. *Organizational Behavior and Human Performance*, **22**, 17–44.
PENNINGTON, N. (1982). Cognitive components of expertise in computer programming: A review of the literature. *Technical Report No. 46*, University of Michigan.
REILLY, R. *et al.*, (1975). In *The Use of Expert Judgment in the Assessment of Experimental Learning*. CAEL Working Paper No. 10.
SHEIL, B. A. (1981). The psychological study of programming. *Computing Surveys*, **13**, 101–120.
SHEPPARD, S. B., CURTIS, B., MILLIMAN, P. & LOVE, T. (1979). Modern coding practices and programmer performance. *Computer*, **12**, 41–49.
SHNEIDERMAN, B. (1980). *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, Massachusetts: Winthrop.
SHROBE, H. E. (1979). Dependency directed reasoning for complex program understanding. *Unpublished Doctoral Dissertation*, Massachusetts Institute of Technology.
SMITH, H. T. & GREEN, T. R. G., Eds. (1980). *Human Interaction With Computers*. London: Academic Press.

SIMON, D. P. & SIMON, H. A. (1978). Individual differences in solving physics problems. In Siegler, R. S., Ed., *Children's Thinking: What Develops?*, pp. 325–348. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
VESSEY, I. (1984). The psychological processes underlying the debugging of computer programs. *Unpublished Doctoral Dissertation*, University of Queensland.
VESSEY, I. & WEBER, R. (1984). Research on structured programming: An empiricist's evaluation. *IEEE Transactions on Software Engineering*, *SE-10*, 397–407.
VITALARI, N. P. (1981). An investigation of the problem solving behavior of systems analysts. *Unpublished Doctoral Dissertation*, University of Minnesota.
VITALARI, N. P. & DICKSON, G. W. (1983). Problem solving for effective systems analysis: An experimental exploration. *Communications of the ACM*, **26**, 948–956.
YOUNGS, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies*, **6**, 361–376.

Appendix A. Discussion of the problem solution

When a program is in error, the error is often manifested in output that differs from the expected. Reference to the correct and incorrect outputs produced in this study (Figs 2b, c) reveals the problem to be one of control over printing certain report fields. In the version with the high-level bug, the branch, salesperson and customer numbers are repeated following the first change in branch number. In the version with the low-level bug, the customer number is repeated following the first change in customer number. Since the program with the high-level bug produces a greater number of erroneous output fields, that problem may appear more difficult at first. However, as already indicated the error is equivalent in both program versions, the difference in output being due to the hierarchical nature of the COBOL code.

Control over changes in each of the three report fields is exercised in two ways: first, by establishing "previous" numbers to test whether a change has taken place; second, by means of a change flag that permits printing of the correct fields. Figure 1 shows the program modules principally responsible for the control functions. (The modules that handle a salesperson change and a branch change are essentially similar to the module B010-PROCESS-CUSTOMER-CHANGE.) "Previous" numbers are initialized with the values in the first input record by the module A000-CREATE-SALES-REPORT on the first execution pass. In module A001-PROCESS-AND-READ, tests are made to determine whether there has been a change in any of the report fields (lines 255, 261 and 266).† If a change has occurred, subordinate modules B010-PROCESS-CUSTOMER-CHANGE, B020-PROCESS-SALESMAN-CHANGE, and B030-PROCESS-BRANCH-CHANGE are called as required to execute the necessary processing. These modules reset the "previous" numbers with current values to prepare to test the next input record (e.g. see line 344 in B010-PROCESS-CUSTOMER-CHANGE). On return to A001-PROCESS-AND-READ, the appropriate change flag is set to 'YES' (lines 259, 264, and 268).‡ In module B000-PROCESS-DETAIL RECORDS, the print module, tests are made on the change flags (lines 295, 301, and 306). If a change has occurred, the relevant input data items are moved to the corresponding report fields, the relevant change flag is reset to 'NO' (lines 299, 304, and 308), DETAIL-LINE is written (lines 315, 316), and finally SPACES are moved to DETAIL-LINE (line 319). A possible solution follows.

† The tests are hierarchical (from branch to customer) and mutually exclusive; that is, a change in branch will also cause processing for change in salesman and change in customer to occur.
‡ Change flags are initialized to 'NO' in the WORKING-STORAGE SECTION.

(1) Ascertain the problem. Note that processing proceeds normally until there is either a branch change (high bug) or a customer change (low bug).

(2) Examine the output file in the file section of the data division (line 55). Note that the output record is defined simply as PIC X(132).

(3) Search for a DETAIL-LINE in the WORKING-STORAGE SECTION. Note the fields in error: BRANCH-NO-REPORT, SALESMAN-NO-REPORT and CUSTOMER-NO-REPORT, or CUSTOMER-NO-REPORT alone.

(4) Find where the DETAIL-LINE is printed: module B000-PROCESS-DETAIL-RECORDS. Check backwards to ascertain where input values are moved to output fields. Check conditions for moving values into the output fields. Note that this occurs when a particular change flag equals 'YES'.

(5) Hypothesize that the change flag always equals 'YES' after the first change is processed because 'NO' is not being moved back to the flag following processing of the change.

(6) Ascertain where 'NO' should be moved to the change flag. Note, there is a definite pattern of movements of 'YES's and 'NO's to the change flags in modules A001-PROCESS-AND-READ and B000-PROCESS-DETAIL-RECORDS respectively.

This is a parsimonious approach to debugging the program: it formulates a hypothesis about the possible cause of error in a logical manner—without making guesses about program structure. The results show it is highly unlikely that debugging will be achieved in this fashion as the programmer will generally need to know more about the program structure before finally deciding on the error.

Subjects frequently proposed two competing hypotheses. The first is that "previous" numbers are not being reset following a change (i.e. subjects have not examined modules B010-PROCESS-CUSTOMER-CHANGE, B020-PROCESS-SALESMAN-CHANGE, and B030-PROCESS-BRANCH-CHANGE sufficiently closely). If this were so, in module A001-PROCESS-AND-READ on every occasion except the first, the "input" number would not be equal to the "previous" number and changes would be processed producing continual total lines. This is *not* the situation presented. The second hypothesis relates to clearing the DETAIL-LINE (or some part of it) before processing the next record (subjects have not examined module B000-PROCESS-DETAIL-RECORDS sufficiently closely). If SPACES were not being moved to DETAIL-LINE, the first part of the report (up to the first change) would not have been printed correctly. In proposing either of these changes, subjects have failed to characterize the problem fully. They generally search for the statements they believe to be absent rather than reasoning about what the situation would be if that were, in fact, the case.

Some inefficiency in debugging COBOL programs occurs because unnecessary references are made to the DATA DIVISION; in particular, in this case, to the WORKING-STORAGE SECTION. One item commonly checked is the initial value of the change flags. Since the first part of the report is correct (i.e. as far as the first customer change or the first branch change), there is no need for programmers to know what values they contain initially.

Int. J. Man-Machine Studies (1985) **23**, 495–525

A knowledge acquisition program for expert systems based on personal construct psychology

JOHN H. BOOSE

Artificial Intelligence Center, 7A-03, Boeing Computer Services, Seattle, Washington 98124, U.S.A.

(*Received 20 March 1985*)

Retrieving problem-solving information from a human expert is a major problem when building an expert system. Methods from George Kelly's personal construct psychology have been incorporated into a computer program, the Expertise Transfer System, which interviews experts, and helps them construct, analyse, test and refine knowledge bases. Conflicts in the problem-solving methods of the expert may be enumerated and explored, and knowledge bases from several experts may be combined into one consultation system. Fast (one to two hour) expert system prototyping is possible with the use of the system, and knowledge bases may be constructed for various expert system tools.

1. Background

1.1. EXPERT SYSTEMS AND KNOWLEDGE ACQUISITION

An *expert system* is a computer system that uses the experience of one or more experts in some problem domain, and applies their problem-solving expertise to make useful inferences for the user of the system. This knowledge is typically gathered in the form of rules of thumb, or *heuristics*. Heuristics enable a human expert to make educated guesses when necessary, to recognize promising approaches to problems, and to deal effectively with incomplete or inconsistent data. Gathering, representing, and using such knowledge in working systems is referred to as *knowledge engineering*. Acquiring such knowledge from the expert is usually the central task in building an expert system.

Getting an expert to articulate problem-solving knowledge is one of the main problems in building expert systems. A long series of incremental interview, build and test cycles are necessary before a system achieves expert performance. The time typically required to build an expert-level prototype is from 6 to 24 months (Buchanan *et al.*, 1983; McDermott, 1982, 1984).

Figure 1 shows our methodology for knowledge elicitation, testing, combination and expert system delivery. Knowledge is elicited from information sources, and placed into an information base. From there, it is analysed and organized into knowledge bases which are directly used with expert system tools such as KS-300™† (an extended version of EMYCIN) and OPS5. These tools are used for rapid prototyping and to test the knowledge for necessity and sufficiency. Test case histories are built up as the knowledge base is incrementally refined. Individual knowledge bases are then combined into knowledge networks, where further testing occurs. Finally, delivery systems are

† KS-300, S.1, and M.1, mentioned in this article, are products of Teknowledge, Inc., of Palo Alto, California.