Int. J. Man–Machine Studies (1993) **39**, 793–812

Characteristics of the mental representations of novice and expert programmers: an empirical study

SUSAN WIEDENBECK AND VIKKI FIX

Computer Science and Engineering Department, University of Nebraska, 115 Ferguson Hall, Lincoln, NE 68502, USA

JEAN SCHOLTZ

Computer Science Department, Portland State University, P.O. Box 751, Portland, OR 97207, USA

(Received 2 June 1992 and accepted in revised form 4 June 1993)

This paper presents five abstract characteristics of the mental representation of computer programs: hierarchical structure, explicit mapping of code to goals, foundation on recognition of recurring patterns, connection of knowledge, and grounding in the program text. An experiment is reported in which expert and novice programmers studied a Pascal program for comprehension and then answered a series of questions about it, designed to show these characteristics if they existed in the mental representations formed. Evidence for all of the abstract characteristics was found in the mental representations of expert programmers. Novices' representations generally lacked the characteristics, but there was evidence that they had the beginnings, although poorly developed, of such characteristics.

1. Introduction and objectives

Comprehension is an important part of programming because other tasks which programmers do are based on it. Jeffries (1982), Gugerty and Olson (1986), and Nanja and Cook (1987) present data which show that program comprehension is crucial to debugging. Holt, Boehm-Davis and Schultz (1987) argue that the ability to abstract information from code is essential to performance in modification tasks. Moreover, even independent of its major role in practical programming tasks, program comprehension may be of value in understanding skilled performance in procedural domains.

Our particular interest is in the mental representation of a program which is formed during program comprehension. The mental representation is the understander's internal model of the target program. It is built up from study of the program text and other information which the understander has available, such as specifications and documentation. The mental representation contains the information extracted from the program during comprehension, subject to some organization imposed by the understander. The mental representation is used to guide tasks of program modification and debugging.

In this paper we are concerned with the characteristics of the mental representation that expert and novice programmers form while studying a program for comprehension. Knowing what kind of information understanders have at their

0020-7373/93/110793 + 20$08.00/0

disposal and how it is characteristically organized is crucial to understanding mental representations and predicting performance on comprehension-related programming tasks. Some studies (Letovsky, 1986; Pennington, 1987*a*, *b*) have addressed the knowledge gained and the mental representations formed by expert programmers during comprehension. Other work (Corritore & Wiedenbeck, 1991) has discussed novice program understanding. There is also some work (Adelson, 1985*a*; Holt *et al.*, 1987) comparing program comprehension and mental representations by expert and novice programmers directly, in the same setting and using the same materials. The comparison of novices and experts is important because the differences found in a direct comparison help to define the contribution of expertise to task performance.

The present study follows up on the work of several previous researchers and concentrates on identifying empirically some characteristics of the mental representation of computer programs with special attention to how experts and novices differ. We propose that an expert's mental representation exhibits five abstract characteristics:

1. It is hierarchical and multi-layered;
2. It contains explicit mappings between the different layers;
3. It is founded on the recognition of basic patterns;
4. It is well connected internally;
5. It is well grounded in the program text.

We propose that novices' mental representations of programs can be shown to lack most of these characteristics.

The following section explains the five characteristics and reviews research related to them. Section 3 presents our methodology. Following that, in Section 4, are the results of the study. Finally, in Section 5, we discuss the state of knowledge about mental representations by expert and novice programmers.

2. Abstract characteristics of a mental representation

Some past research on program comprehension has focused on the kinds of information that programmers extract during program comprehension and which thus becomes part of their mental representation of a program. The focus of this past research has been to determine whether programmers build a representation consisting mostly of concrete information about *how* the program works or of functional information about *what* the program does. Adelson (1985*a*) compared novices' and experts' mental representations of small programs and found that novices develop a representation consisting of concrete information, while functional information is more prominent in experts' representations. In her study novices surpassed experts in answering certain concrete questions, apparently because such information was more readily available to them in their mental representations. Pennington (1987*a*, *b*) argued that the mental representation of a program is made up of many different kinds of information that exist simultaneously in a program text. She divided information in programs into five categories: operations, control flow, data flow, state, and function. Although she considers a finer breakdown of information into categories, her results are similar to Adelson's. Among professional programmers, better comprehenders were distinguished from poorer comprehenders

by their mastery of functional information. However, she did observe that poorer comprehenders improved their functional understanding after doing a modification task.

In the research reported here we do not focus on the categories of information making up the content of the representation but rather on the representation's general features. These are features which characterize the representation as a whole. We investigated a group of features which had been suggested in the programming literature as characteristic of mature mental representations in programming: hierarchical multi-layered structure, explicit mappings, incorporation of basic recurring patterns, well-connectedness, and foundation in the program text. We call these *abstract features* of a representation to distinguish them from the categories of information that form the content of the representation, as studied by Adelson and Pennington. In our view these features are manifestations of general comprehension mechanisms used in program understanding, and they help to explain at a deeper level why certain categories of information are extracted by programmers during the comprehension process. They also give insights about the mechanisms which humans use to organize complex, detailed procedural information. Each of these features is described below with references to related work.

A **hierarchically structured** mental representation of a program is one which conceptualizes elements of a program as forming a layered network of arbitrary depth and breadth, depending on the specific program (Letovsky, 1986). In such a representation the nodes represent goals and subgoals to be achieved, while the links represent the connections between them. Thus, a goal at one level of the network is broken down into the subgoals that make it up at the next level. This kind of hierarchical organization of knowledge may be fairly explicit in a program's code to the extent that the program is developed following common software engineering guidelines of modularity, top-down decomposition, and stepwise refinement in program design (see e.g. Pressman, 1992; Sommerville, 1992). However, it may be only implicit in programs which are not modularized or are poorly decomposed into modules. A hierarchical structure of mental representations by programmers has been suggested by some empirical studies. Both Jeffries (1982) and Nanja and Cook (1987) observed that advanced programmers (but not novices) used a strategy of reading a program in the order in which it would be executed, and they believed that this led to development of a hierarchical understanding of the program. However, Holt *et al.* (1987), using a free recall technique, found no significant difference between novices and experts in the depth of their mental models of programs. We hypothesized that expert programmers would show more evidence of hierarchical structure in their mental representations of a program than would novices.

An important feature of the mental representations is not just the existence of multiple layers of representation but the existence of **explicit mappings** between the layers. Letovsky (1986) has argued that the highest level of representation of the program, the specification, consists of its overall goals and is usually readily understandable to programmers from such sources as mnemonic names used in programs or documentation. Likewise, the lowest level, the implementation, which consists of the data structures and actions of the program, is also readily understandable, i.e. a programmer can understand the action of each line of code in

isolation. The problem in comprehension is building a bridge between the specification and the implementation. Letovsky calls this bridging process assimilation and says that it consists of filling in purpose links which map the implementation to the specification through a series of layers. This conceptualization of the comprehension process filling in purpose links fits with Brooks's (1983) theory that program comprehension should be seen as creating a mapping between high-level goals and their code representation. In empirical work, Pennington (1987*a*) found that the most skilled expert programmers tied their hypotheses about a program's function to specific information in the program code itself. This was not true of some less skilled experts who made numerous, speculative hypotheses about a program's function from triggers like variable names, without ever really verifying them by figuring out what the segments of code did. We expected a strong difference between novices and experts in their ability to link specific segments of code to program goals. However, we did not expect a correspondingly strong difference in their ability to understand overall program goals.

The use of **recurring basic patterns** as a foundation for knowledge representation has been a theme of programming research for a number of years. The idea is most closely associated with Soloway and his colleagues (Soloway, Ehrlich, Bonar & Greenspan, 1982; Ehrlich & Soloway, 1984; Soloway & Ehrlich, 1984). According to Soloway, programming knowledge is represented as a set of frame-like structures, called plans, for handling stereotypical situations which arise frequently in programming. Thus, a programmer would possess plans in memory for operations such as looping, counting, accumulating values, etc. These basic plans would serve as the building blocks for writing programs and also as the knowledge base for comprehending programs. If basic programming knowledge is stored as plans, then one would expect to see evidence of plan recognition during program comprehension and plan-like structures in the mental representation of individual programs. Soloway and Ehrlich (1984) presented evidence that experienced programmers' comprehension was disrupted by programs that were written in an unplan-like way, thus supporting the idea that plan recognition must occur in comprehension. Moreover, they found that subjects who had seen a subtly unplan-like program tended to recall seeing the plan-like program instead, thus suggesting that plan structures formed a part of their mental representation. We hypothesized that there would be a difference between novices and experts in their ability to connect program code with plan labels for all but the very simplest plans.

A **well-connected** representation is one where the programmer understands how parts of the program interact with one another. Interactions are difficult to understand, probably because they embody instances of delocalized plans (Soloway, Pinto, Letovsky, Littman & Lampert, 1988). Delocalized plans are programming plans, as described above, but they are not implemented in a contiguous body of code, such as a single module. Rather the code implementing them is scattered throughout the program. This lack of contiguity introduces difficulties in comprehension, since one cannot see the plan implementation as a single coherent unit. Jeffries, Turner, Polson and Atwood (1981) found that experienced programmers designing a program pay special attention to parts of the code that interact, for example the design of interfaces between related modules. Given the difficulties posed by the interaction of segments of code, we expected that experienced

programmers would try to extract such information in the comprehension process and it would thus be included in their mental representation of the program. Novices, on the other hand, would be unlikely to concentrate on this type of information.

Representations which are **well grounded** in the program text are also suggested in the literature as characteristic of good mental representations. Well-grounded representations include specific details about the locations where structures and operations occur in the program. An understanding grounded in the program text is useful because it allows programmers who have studied a program once to relocate with minimal search information which is needed to carry out programming tasks. In her protocols, Jeffries (1982) observed that expert programmers were very skillful at locating information which they had previously seen in a program when it was needed a second time. Novices did not seem to know where to look for information and would search either randomly or else do a linear search from the beginning of the program. We tested experts' and novices' ability to locate information in a program text with the hypothesis that experts would surpass novices at locating most, but not all, kinds of information.

Method

OVERVIEW

Novice and expert programmers studied a 135 line Pascal program and then answered questions about it. The questions were designed to investigate the five abstract characteristics of the mental representation described earlier. The questions about the abstract characteristics were each supplemented by questions which asked for similar information but in a context not involving the abstract characteristic. These latter questions were included to help us to judge whether any group differences observed reflected differences with respect to the abstract characteristics or merely differences in the total amount of programming knowledge of the two groups.

SUBJECTS

Twenty novice and 20 expert programmers took part in the study. All of the subjects were volunteers. They received $5·00 for their participation. The novice programmers were undergraduate students who had recently completed a first semester Pascal course at the University of Nebraska, the University of South Dakota, or Portland State University. In addition to one university-level Pascal course, 17 subjects had taken a BASIC course in high school or college, three had taken a high school Pascal course, one had studied FORTRAN, and one an assembly language. None of the novices had any work experience as programmers, and none had taught programming.

The expert subjects were professional programmers recruited in Oregon and Nebraska. The full-time professional programming experience of these subjects

ranged between 2–13 years. The mean amount of experience was 7·7 years and the median was 7 years. Six subjects had Masters degrees in computer science, 10 had Bachelors degrees in computer science, two had Bachelors degrees in other fields, and two had no degree but had computer science course work in college or a technical school. Twelve subjects had taught programming courses, most often FORTRAN or C. Nineteen subjects had written or maintained large programs, defined as over 3000 lines of code, in one or more language. Seventeen subjects had written over 100 programs in at least one language. Although the professional programmers differed from each other in the amount and type of their programming experience, they were very clearly distinct from the novice group, based on the demographic data that we collected.

MATERIALS

The program used in this study is shown in full in Appendix 1 and a summary of it is shown in Figure 1. The program is written in Pascal. It is 135 lines long and occupied 3 pages as printed out for use in the study.

The program manipulates student data. For each student it reads in a student identification number and a score from a file and stores them in parallel arrays. Then it sorts the arrays by identification number using a selection sort. It gives the program user a chance to interactively change a score. If the user wants to change a score, the student identification number is located using a binary search, after which the user enters the new score. The program then computes the average and maximum scores of the group of students and prints them out. Finally, it prints out to the screen the identification numbers and scores, and it also writes them to an output file.

The program consists of a main program and nine subprograms. The deepest nesting of program blocks is four levels deep. The operations contained in the program had been covered in the beginning computer science courses taken by the novices: reading and writing files, interactively reading and writing to the terminal, manipulating arrays, sorting, searching, finding an average, and finding a maximum value. The use of procedures and functions, parameter passing, local variables, and the nesting of subprograms were all taught. Also, the context of working with student identification numbers and scores was familiar to the novices as well as to the experts. Mnemonic identifiers were used in the program, which was also

```
Program DataAnalysis
    Procedure ReadData–Reads IDs and scores into arrays, counts number of inputs
    Procedure ProcessData–Calls procedures to sort input, change scores, compute statistics
        Procedure OrderData–Sorts input data by student ID using selection sort
        Procedure ChangeScores–Calls search procedure to locate a student ID and makes change
            Procedure FindData–Searches for a particular student ID using binary search
        Procedure ComputeStats–Calls functions to compute average and maximum scores
            Function Average–Computes average score
            Function Max–Computes maximum score
    Procedure RewriteData–Writes data with changes to screen and to a new file
```

FIGURE 1. Outline of program modules with annotations.

TABLE 1
Summary of comprehension questions

Question number	Question content	Abstract category	Significant difference
1	Match procedure names to the procedures they call	Hierarchical structure	$p = 0{\cdot}0043$
2	List procedure names	Hierarchical structure	N.S.
3	Write description of goals of selected procedures	Explicit mappings	$p = 0{\cdot}0001$
4	Write description of principal goals of program	Explicit mappings	N.S.
5	Label complex code segments with plan label	Recurring patterns	$p = 0{\cdot}0001$
6	Label simple code segments with plan label	Recurring patterns	N.S.
7	List names used for same data objects in different program units	Well connected	$p = 0{\cdot}0013$
8	List important variable names	Well connected	N.S.
9	Fill in names of program units in a skeleton outline of the program	Well grounded	$p = 0{\cdot}0007$
10	Match variable names to the procedures in which they occur	Well grounded	$p = 0{\cdot}0006$
11	Indicate physical location of invariant program parts	Well grounded	N.S.

indented in a normal style to show nesting of blocks and of individual statements inside compound statements. No documentation was given.

The comprehension questions which subjects answered are described in Table 1. Note that the questions are arranged in Table 1 to facilitate a comprehensible discussion, and this is not the order in which subjects saw them. The questions focused on determining whether programmers exhibited the abstract characteristics in their representations of the program. The correspondence of the questions to the abstract characteristics is discussed below. The different questions relevant to a given abstract characteristic were balanced to require a similar amount of information from the subject (e.g. Questions 5 and 6, which were both relevant to the recurring patterns characteristic, each required an answer consisting of 4 parts).

1. Hierarchical, multi-layered structure

Questions 1 and 2 were designed to elicit whether the subjects' mental representations had a hierarchical, layered structure. The stimulus program reflected a clear hierarchical, layered structure in its use of nested procedures and functions. There was a top-level main program, three second-level procedures called by the main program (ReadData, ProcessData, RewriteData), three third-level procedures nested inside the ProcessData procedure (OrderData, ChangeScores, ComputeStats), and finally two fourth-level functions to do the computations within

ComputeStats (Average and Max). Question 1 asked subjects to match procedure names to the procedures which they called. A high score on this question would indicate that a subject had understood the hierarchical structure of the program and made it a part of the mental representation.

Question 2 was used to judge whether a difference in matching the calling and called units in the previous question was likely to be an artefact of better overall memory for programming materials as opposed to hierarchical structure of the representation. Question 2 required subjects simply to list the names of the procedures used in the program, without regard to their physical order or the calling sequence. Thus, this question dealt with the same program elements, procedure names, but independent of their hierarchical context. A lack of difference between groups on this question would suggest that a difference on Question 1 was not explained merely by overall better memory of program elements.

2. Well-developed mapping of code to goals

Questions 3 and 4 were relevant to judging the ability to link code to program goals. Question 3 asked the subject to write a brief description (a few sentences) about two procedures in the program, FindData (the binary search) and ComputeStats (the procedure which controlled the numerical computations). Subjects were instructed to tell what program goals the procedures realized. Thus, it was a measure of the subjects' ability to map between the code and the program goals. Subjects were also asked to write a sentence or two about *how* the procedure carried out its goals. This was included to gather information about whether subjects' mental representations also contained information about methods by which goals were implemented.

Question 4, by contrast, asked the subject to write a brief description of what the whole program did, including the main goals. In asking about only the high-level goals, Question 4 involved information which was likely to be available to programmers superficially, without a need for detailed understanding of what code implemented the various goals or how. It was expected that both novice and experienced programmers would be able to extract the overall goals asked for in this question but that there would be group differences in the ability to map between high-level goals and the program code, as required by Question 3.

3. Recurring patterns

Questions 5 and 6 looked for evidence of the incorporation of basic plan knowledge in the understanding of the program. In each question the subject was given brief code segments from the program and was asked to give a label to each segment, consisting of a few words, to tell what it did (e.g. "initializes variable"). The patterns in Questions 5 and 6 were of different levels of complexity. Question 5 contained somewhat complex patterns, including the linear search for the largest value in the array, the segment which read input in a loop until end of file and counted the elements read, the sort routine, and the binary search routine. These segments were between 3–9 statements long and contained several actions which worked together to achieve their goal. For example, the segment of code for finding a maximum value, which was the simplest of the four segments, included looping, testing a value and storing a value.

Question 6 also contained stereotyped patterns but ones of the most elementary kind. These segments were either one or two statements long and included the code used to average two values, increment a counter, sum an array, and write out an array. They contained no more complex control structures than the simple FOR-loop used in summing an array. Being able to label any of the code segments in Questions 5 or 6 with an appropriate plan label is evidence of the use of plan knowledge in the understanding of the program. However, the different levels of complexity of the patterns in the two questions was meant to distinguish the ability of programmers at different skill levels to bring plan knowledge to bear in representing the code.

4. Well-connected representation

Questions 7 and 8 were relevant to judging the well-connectedness of the representation. Question 7 gave subjects a list of variable names and asked them to indicate what other names, if any, those variables were known by in different program units. Thus, to answer this question subjects had to have an idea of what a data item represented, and they had to understand how it was passed through the program, possibly with different identifiers, while still representing the same object. Question 8 asked subjects to list the names of all variables in the program. However, we were interested in just 7 variables which were used for the principal data elements in different units of the program. These 7 variables were: ID, Score, NumStudents, a, a1, a2, and count. Generating these could indicate knowledge of the essential objects in the program, but it would not indicate well-connectedness in the representation.

5. Well grounded in the text

Questions 9, 10 and 11 were used to judge how well grounded the representation was in the program text. Question 9 presented a list of the names of the program units (nine in all) and beside that a skeletal template of the program which represented the location of the program units by boxes. Nested units were shown by a box within a box. Subjects had to write the given subprogram names in the proper boxes in the template. High performance on this task would be an indication that the subject had synthesized an overview of the location of procedural units and incorporated it in the mental representation. While Question 9 had to do with the location of actions in the text, Question 10 concerned the location of objects. Question 10 required subjects to match different variable names to the program units in which they appeared (including some names which occurred in more than one unit). The ability to do this would also show well-groundedness because the subject had a representation of where objects occurred in the text. The last question asked the subject to indicate the location of certain elements in the program text which are invariant in that they can be described by a relative location which is unchanging from one program to the next (e.g. Question: "Where is the end statement of the main program located?"—Answer: lst line of text; Question: "Where were the declarations for local variables of the procedures located?"—Answer: after the procedure heading). Correct answers on Question 11 would show some ability to handle the program as a piece of text having a particular structure,

based on syntactic knowledge. However, it would not show program-specific grounding of knowledge in the program text.

PROCEDURE

Subjects were run individually or in groups of two or three. The subject first filled out a background questionnaire on programming experience. Then the subject was given a listing of the program to study for 15 min. Pretesting had shown that 15 min was enough for both novices and experts to study the program to their own satisfaction. Subjects were told to study the program in detail so that they understood everything that happened in it. They were allowed to mark the program listing if they wished.

At the end of the study period the program listing was taken away. Subjects were given a question booklet with one question per page. The questions were not arranged in the numerical order shown in Table 1 but in an order chosen so that the answer in one question did not give away the answer to a later question. For example, the questions asking subjects to generate subprogram names or variable names were placed before matching questions which would reveal the names. Subjects were told to answer the questions in order. They were not allowed to return to a previous question once they had turned the page. Subjects were allowed to take as long as they needed to work through the question booklet. For experts the times ranged from 40–75 min with the average around one hour. For novices the times ranged from 45–90 min with the average around 75 min.

Data analysis and results

SCORING

The performance on the comprehension questions was evaluated by two independent judges. Each judge scored the correctness of the responses to all questions. Uniform scoring criteria were developed and were used by the judges. Only the questions which asked the subject to write a description of some segment of the code required subjective judgement by the judges. The level of agreement of the judges on these questions was 97%. As the scoring criteria below indicate, a $\frac{1}{2}$ point penalty was subtracted for wrong answers on four of the 11 questions. On these four questions there was a very large number of wrong answers. The wrong answers may have come from two sources: (1) wrong information in the mental representation, proceeding from misunderstanding of the program; and (2) guessing, when the subject was simply lacking the required information in the mental representation. In our view, having wrong information in the mental representation is more serious than wrong guesses and ought to be penalized just as strongly as correct information is rewarded. However, it was impossible to distinguish wrong information from guessing in the results. Thus, a conservative approach was taken to penalizing wrong answers, deducting $\frac{1}{2}$ point rather than a full point. Outside of the four questions indicated below, subjects were not penalized for giving incorrect answers. The errors subjects made on the other seven questions were almost entirely the failure to supply requested information, rather than supplying erroneous information.

Scoring criteria

Question 1: Matching procedure names to the procedures they called. One point was given for each correct call. To adjust for guessing, $\frac{1}{2}$ point was subtracted for each incorrect call indicated.

Question 2: Recall of names of procedures. One point was given for each name that matched exactly. Capitalization was ignored.

Question 3: Description of goals implemented by subprograms. A list of required description elements was developed by the authors and one point was given for each element present.

Question 4: Description of overall program goals. The same procedure was used as for Question 3.

Question 5: Labelling moderately complex plans. One point was given for each correct label. If the subject simply restated in English what each line in the segment did, as opposed to giving it a plan label, it was considered incorrect because it did not show plan knowledge.

Question 6: Labelling very simple plans. The same procedure was used as for Question 5.

Question 7: Providing names used for a data object in different program units. The score was calculated as the number of correct names minus $\frac{1}{2}$ the number of incorrect names.

Question 8: Recall of principal variable names. One point was given for each name that matched exactly. Capitalization was ignored.

Question 9: Filling in names of program units on a template. One point was given for each name placed in the correct slot of the template.

Question 10: Matching variable names to the subprograms in which they occurred. The score was calculated as one point for each correct match of a variable to a subprogram minus $\frac{1}{2}$ point for each incorrect match.

Question 11: Description of location of invariant elements of the program. One point was given for each correct description.

RESULTS

First a MANOVA was run to test whether there was an overall difference in performance between experts and novices. It was found that the experts scored significantly higher than the novices ($F(11, 28) = 8{\cdot}9635$, $p = 0{\cdot}0001$). Following the significant MANOVA, individual ANOVA procedures were run on the different questions. For these ANOVAs on the individual performance variables the alpha level was set at 0·0045, i.e. 0·05 divided by 11, the number of tests performed. This was done to reduce the likelihood of Type I errors, i.e. rejecting null hypotheses that were true. The results of the individual ANOVAs are presented in the following paragraphs.

1. Hierarchical, multi-layered structure

Question 1 tested the presence of hierarchy in the subjects' representations. Experts scored significantly higher than novices ($F(1, 38) = 9{\cdot}20$, $p = 0{\cdot}0043$). The experts' mean score was 6·03 out of a possible score of 7 (if the subject had all the correct calls asked for and no spurious ones). The novices' mean was 3·40. Question 2

tested knowledge about the same program units as Question 1 but without requiring an understanding of the hierarchical structure. The ANOVA was not significant. The experts' mean score was 4·25 out of a possible score of 7, and the novices' mean score was 2·85.

2. Well developed mapping of code to goals

Question 3 asked for information about how two subprograms fulfilled goals of the program. The subjects' descriptions were analysed for the presence of seven specific information elements. Experts scored significantly higher than novices on Question 3 ($F(1, 38) = 11{\cdot}87$, $p = 0{\cdot}0014$). The experts' mean out of the 7 elements was 5·2, while the novices' mean was 2·8. Although our main interest was subjects' ability to link code to program goals, we also analysed subjects' statements about the methods which the subprograms used to achieve their goals. For this analysis, subjects' descriptions were scored for the presence of four specific information elements related to method. Experts also scored significantly higher on this measure ($F(1, 38) = 34{\cdot}45$, $p = 0{\cdot}0001$). The experts' mean was 2·90 out of the 4 elements, and the novices' mean was 0·85. Question 4 asked only about the overall goals of the program. There was no significant difference between novices and experts on Question 4. The experts' mean was 6·15 and the novices' mean was 5·50 out of 7 elements on which the descriptions were scored.

3. Recurring patterns

Question 5 contained moderately complex recurring patterns, while Question 6 contained very simple recurring patterns. The results showed that there was a significant difference on Question 5 ($F(1, 38) = 57{\cdot}58$, $p = 0{\cdot}0001$). Out of a possible score of 4 the experts' mean was 3·7 and the novices' mean was 1·7. There was no significant difference between novices and experts on Question 6. Out of a possible score of 4 the experts' mean was 4 and the novices' mean was 3·7.

4. Well-connected representation

Question 7 probed about knowledge of data connections by asking what were the names used for the same conceptual objects in different subprograms. The experts were superior to the novices ($F(1, 38) = 11{\cdot}99$, $p = 0{\cdot}0013$). The experts' mean was 3·55 out of a possible score of 7, and the novices' mean was 1·43. Question 8 simply asked for a listing of names of data elements, independent of any connections to other data names used in the program. There was no significant difference between the two groups on the seven principal variables. The experts' mean was 4·80 out of 7 and the novices' mean was 3·90.

5. Well grounded in the text

There was a significant difference on Question 9, the location of subprogram names on the program template ($F(1, 38) = 13{\cdot}58$, $p = 0{\cdot}0007$). The experts' mean was 8·8 out of 9 names, and the novices' mean was 6·7. Question 10 involved linking variable names to the context in which they appeared. The experts performed significantly better than the novices ($F(1, 38) = 14{\cdot}08$, $p = 0{\cdot}0006$). The experts' mean was 5·63 out of a possible score of 10, while the novices' mean score was 1·65.

Question 11 asked about locations of elements in the code that have a fixed absolute or relative location. There was no significant difference on this measure. The experts' mean was 9·3, and the novices' mean was 9·1 out of 10.

Discussion

The results of this study tend to support the existence of the five abstract characteristics in the mental representations of expert programmers. However, novice programmers do not show the same characteristics in their mental representations, or do not show them to the same degree.

A program is a procedural text which expresses actions. Although it is written as a linear sequence of statements, the underlying structure is a hierarchy in which an action is linked to other actions that are dependent on it. The hierarchical, multi-layered nature of program representation had been suggested by some researchers but not supported by others. This study supported the hierarchical character of expert program representations. Experts were able to discern the hierarchical structure of a program, as represented in the calling sequence of the subprograms, and they were able to recall or reconstruct it when needed. Novices were very poor at this task, although they were not significantly different from experts in their ability to remember the subprogram names, when they were queried about them independently of their hierarchical relationships. From this we are led to conclude that the experts' better memory for programs is not entirely quantitative (although it may be partly quantitative), but also has to do with the nature of the program elements being recalled. We suggest that experts' memory for hierarchical relations is better because they recognize the procedural nature of a program and thus selectively seek the non-linear relationships among actions, essentially performing a sort of rough simulation in reading a program. Although novices remember some kinds of program information quite well, they apparently do not have a strategy of selectively searching out the hierarchical sequence of actions. It is possible that they do not fully distinguish a program text from more familiar linearly-structured prose text.

We speculate that the hierarchical structure of the mental representation can be strongly influenced by the nature of the program itself. The program we studied had a clear, consistent hierarchy that was readily discernible in the calling sequence of the modules. In a program where the hierarchical relationships were less clearly emphasized in the code we believe that expert representations would be more similar to novice representations.

This experiment also confirmed that expert programmers make explicit mappings between segments of program code and the subgoals that they implement. Experts and novices did not differ from each other in understanding the overall goals and major subgoals of the program, things which they may have been able to discern from superficial information, such as identifiers and literal messages embedded in the program. However, when it came to understanding the mapping between high-level goals and the program code, i.e. which specific segments of code implement the various goals of the program, there was a difference. Experts showed an ability to make a mapping between high-level goals and the individual code segments that implemented them, while novices were not able to do so to a great

degree. The contrasting results on our two questions concerning program goals suggests why novices may sometimes equal experts in understanding program function (e.g. Wiedenbeck, 1986), but fall short other times (e.g. Adelson, 1985*a*).

This result calls to mind Pennington's observation that poorer comprehenders tend to make many speculations about program goals without following through to verify them in the code. Pennington was discussing better and poorer comprehending expert programmers, but it appears that a similar distinction can be made between novices and experts. Similarly, Littman, Pinto, Letovsky and Soloway (1986) observed some expert programmers who used a comprehension strategy in which they consciously decided *not* to try to understand details of the mapping, unless they felt that the details were required to carry out a given task, such as a modification. This strategy often led to failure at the task. In the case of our results, we do not claim that novices made a conscious decision to avoid making a detailed mapping of code to program goals. It is possible that they tried but were unable to do so.

This experiment presents more evidence of the use of basic plans in comprehension. Both novices and experts recognized and gave correct stereotypical labels to the simplest recurring patterns. Expert programmers went beyond this to recognize and label instances of more complex recurring patterns. Furthermore, the experts used the same stereotyped plan label to describe the binary search when referring to it in Question 4. Question 4 required them to write descriptions of parts of the program in their own words. Their generation of the same plan label in this other context indicates that the plan labels probably become part of the mental representation of the program.

This experiment in no way tested the limit of the expert subjects' plan recognition ability. In fact, we expect that they could recognize and label many other, more sophisticated, plans. The experiment does show that novices are lost when they go beyond the very simplest recurring patterns. They failed to correctly label the more complex plans even though they had been exposed to these very plans previously in class. Instead, they gave either no answer at all or else a line-by-line description that was merely a restatement in English of the lines of code. Thus, it appears that from their early stages of development, programmers begin to learn and take advantage of recurring patterns in program comprehension (and one presumes in program composition, too). However, the skill must develop slowly with time and exposure to many programs.

Interestingly, the labels which were given by experts for the complex code segments were at different levels of specificity. This was evident in a comparison of the labels given for the sort and search segments. The sort segment was actually a selection sort, but subjects usually used the label "sort", rather than the more explicit "selection sort". The search segment was a binary search and a large majority of expert subjects did label it specifically as "binary search". The more specific label for the binary search may reflect that programmers perceive the binary search method as being distinctly different from a linear search. On the other hand, in the case of sorting there are many sorts which use similar methods and may be perceived as a single category. This difference in labelling program segments reminds one of Rosch, Mervis, Gray, Johnson and Boyes-Braem's (1976) superordinate-basic-subordinate levels in natural categories (e.g. furniture—chair—

director's chair). Our expert subjects named the sort at the middle level of abstraction but named the binary search at the lowest level, making use of more specific attributes which distinguished it from other searches. Rosch *et al.* observed that people usually name objects at the basic level but that domain experts may name objects at a more specific level. Adelson (1985*b*) showed that a basic level of abstraction of computing concepts exists, corresponding to categories like sort and search. In an experiment dealing with the "horizontal dimension" of categories, she found that the selection sort is the most prototypical exemplar of the category sort. This prototypicality may be an alternate explanation of why our expert subjects gave simply the label sort to a selection sort.

The representations formed by the expert programmers in this study were well-connected internally. Experts were able to link a single conceptual object in the program to the different names it was known by in various program units. This task, at which novices performed poorly, showed a unity of understanding of objects in the program. The question about connectedness dealt with objects and their interrelations. However, it was not the case that experts were simply better at remembering objects. Novices were just as successful as experts at recalling the major program variables when it was not a question of making the connections between objects. Pennington found that on initial study of a program, poorer expert comprehenders were less successful at answering questions on data flow relations. Data flow relations represent the major kind of connections that exist between modules in a program, so this result complements hers in showing that novices have a poor grasp of data connections throughout a program. Experts by contrast seem to seek the relations of objects, which leads to a connected view of the program, spanning its discrete modules.

The experts' representations were well grounded in the program text. Experts were able to tell where various program units were located in the program. They knew the physical structure of the program they had studied. They were also able to tell in what program units various data objects were used. Thus, for experts objects were linked to actions which were linked to physical locations. One would expect experts to be able to quickly carry out tasks requiring locating information in the text, such as finding where an action was carried out or what had been done to some object at a previous point. Novices were poorer than experts at telling the location of modules and objects. They did show some ability to describe the locations of a few program objects that could be predicted on syntactic grounds alone. However, when they had to answer location questions that could only be answered from an understanding of the specific program their performance did not match that of experts. Their understanding was not well grounded in the physical program text. This lack of a mental map of the program could cause great difficulties in complex maintenance or debugging tasks.

Conclusions

In this paper we have described a set of abstract characteristics of the mental representation of programs. We have shown that expert programmers possess these characteristics while novices lack them in the developed form possessed by the experts. These results provide experimental support for some of the characteristics

of novice and expert program comprehension observed by Jeffries (1982) in her protocol analysis. Taken together, these differences in the mental representation may provide a partial explanation of why novice performance is poorer than expert performance on tasks which have program comprehension as a prerequisite. The results suggest that a number of skills contribute to the formation of the mental representation: for example, skill at recognizing basic recurring patterns, skill at understanding the particular structure inherent in a program text, skill at recognizing the links tying the separate program modules together, etc. When a programmer exercises these skills, a good representation which supports comprehension-related programming tasks is likely to emerge.

A limitation of this experiment is the lack of "naturalness" of the task and its possible implications. The subjects were instructed to study the program in order to comprehend in detail its structure and function. While the novices did not consider this an unusual task, apparently some of the experts did. A few of them commented that in studying a program they normally had a concrete objective in mind, such as finding a bug or determining the effects of a potential modification. In this case they were on a fishing expedition and, as a result, were not sure where to focus their efforts. This raises the question of what information experts would extract during program comprehension when given a more concrete goal. We speculate that the objects and relations recalled could change depending on the nature of the situation posed in the instructions. For example, a modification instruction in which the subject was told the modification to be made would be likely to lead to concentration on a specific part of the program at the expense of other parts. Also, the size of the program and the familiarity of its domain would influence the information gathered during comprehension. Thus, with respect to the results reported here, we can only claim that experts are capable of creating a broad, multi-faceted representation, not that they necessarily do so every time they work with a program. However, we do expect that maintenance programmers who work with a large program over a period of time eventually develop a multi-faceted representation similar to what we found.

Another question about this study concerns the validity of the metrics used. The metrics were developed explicitly for this study and, to our knowledge, none of them had been used in previous studies of the cognition of programming. The finding of expert/novice differences suggests that they are, in fact, meaningful for the materials used in this study. However, we would like to validate them formally by varying the characteristics in the program being comprehended (well-groundedness, explicitness mappings, etc.) and determining whether this changes the expert's mental representation. We plan to do this as future work.

One may question why novices do not exhibit the same characteristics in their mental representations as experts. Clearly, some of the difference is simply the result of possessing less programming knowledge. For example, knowledge of recurring patterns may be deficient among novices and need to be built up through study and practice. On the other hand, some other characteristics of expert mental representations are based on information readily available in the program, yet novices do not extract it. Examples are the hierarchical structure inherent in the flow of control of the program and the connections between modulues which are explicit in the passage of data. It may be that novice programmers do not pick up such

information because they are using a different program comprehension strategy than experts. Littman, Pinto, Letovsky and Soloway (1986) describe "systematic" and "as-needed" comprehension strategies. In the systematic strategy the programmer performs symbolic execution of control and data flow paths and thus detects causal interactions of components in the program. This is done before attempting a programming task, such as modification. In the as-needed strategy the programmer tries to minimize study of the program, gathering only information that seems required by the programming task being undertaken. According to Littman *et al.*, the lack of concentration on control and data flow among modules may result in failure to detect causal interactions. It may be that novice programmers use an as-needed strategy, either because of failure to understand the critical importance of the interactions or because of lack of skill at symbolic execution. This could result in lack of hierarchical structure and poorly-connected mental representations.

We believe that to some extent the development of good mental representations may be promoted through instruction. For example, the teaching of a top-down reading strategy would promote the formation of a hierarchical representation of a program. An emphasis in instruction on the interfaces between modules as a source of errors would lead to a better-connected representation. A representation would be better grounded physically in the program text if instruction emphasized the structural elements inherent in a program text and how a program text differs from linear English text. Beyond instruction, good mental representations are likely to be developed as students carry out programming tasks, such as debugging and modification of numerous programs, and learn from a distillation of these experiences what characteristics are important to support programming tasks. Instruction will not replace the role of experience in developing good mental representations but may expedite the process.

References

ADELSON, B. (1985*a*). When novices surpass experts: the difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition,* **10,** 483–495.

ADELSON, B. (1985*b*). Comparing natural and abstract categories: a case study from computer science. *Cognitive Science,* **9,** 417–430.

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

CORRITORE, C. L. & WIEDENBECK, S. (1991). What do novices learn during program comprehension? *International Journal of Human–Computer Interaction,* **3,** 199–222.

EHRLICH, K. & SOLOWAY, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. C. THOMAS & M. L. SCHNEIDER, Eds. *Human Factors in Computer Systems.* Norwood, NJ: Ablex.

GUGERTY, L. & OLSON, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

HOLT, R. W., BOEHM-DAVIS, D. A. & SCHULTZ, A. C. (1987). Mental representations of programs for student and professional programmers. In G. M. OLSON, S. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

JEFFRIES, R. (1982). A comparison of the debugging behavior of novice and expert

programmers. *Paper presented at the American Educational Research Association Annual meeting,* New York.

JEFFRIES, R. TURNER, A. A., POLSON, P. G. & ATWOOD, M. E. (1981). The processes involved in designing software. In J. R. ANDERSON, Ed. *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

LETOVSKY, S. (1986). Cognitive processes in program comprehension. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

LITTMAN, D. C., PINTO, J., LETOVSKY, S. & SOLOWAY, E. (1986). Mental models and software maintenance. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

NANJA, M. & COOK, C. R. (1987). An analysis of the on-line debugging process. In G. M. OLSON, S. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

PENNINGTON, N. (1987*a*). Comprehension strategies in programming. In G. M. OLSON, S. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

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

PRESSMAN, R. S. (1992). *Software Engineering: A Practitioner's Approach,* 3rd ed. New York: McGraw-Hill.

ROSCH, E., MERVIS, C. B., GRAY, W. D., JOHNSON, D. M. & BOYES-BRAEM, P. (1976). Basic objects in natural categories. *Cognitive Psychology,* **8,** 382–439.

SOLOWAY, E. & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **10,** 595–609.

SOLOWAY, E., EHRLICH, K., BONAR, J. & GREENSPAN, J. (1982). What do novices know about programming? In A. BADRE & B. SHNEIDERMAN, Eds. *Directions in Human–Computer Interaction.* Norwood, NJ: Ablex.

SOLOWAY, E., PINTO, J., LETOVSKY, S., LITTMAN, D. & LAMPERT, R. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM,* **31,** 1257–1267.

SOMMERVILLE, I. (1992). *Software Engineering,* 4th ed. Wokingham, UK: Addison-Wesley.

WIEDENBECK, S. (1986). Processes in computer program comprehension. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Paper accepted for publication by Associate Editor Dr Bill Curtis.

Appendix 1: the Pascal program used in the study

```
program DataAnalysis (input, output, infile, outfile);

    const
        size = 1000;
    type
        arraytype = array[1..size] of integer;
    var
        Score, ID: arraytype;
        NumStudents: integer;
        infile, outfile: text;

    procedure ReadData (var a1, a2: arraytype;
                        var count: integer);
    begin
        count := 0;
        while not eof(infile) do begin
            count := count + 1;
            readln (infile, a1[count], a2[count])
        end
    end;
```

```
procedure ProcessData (var ID, Score: arraytype;
                       NumStudents: integer);

    procedure OrderData (var a1, a2: arraytype;
                         count: integer);
        var
            index1, index2, min, temp: integer;
    begin
        for index1 := 1 to count-1 do begin
            min := index1;
            for index2 := index1+1 to count do
                if a1[index2] < a1[min] then
                    min := index2;
            temp := a1[min];
            a1[min] := a1[index1];
            a1[index1] := temp;
            temp := a2[min];
            a2[min] := a2[index1];
            a2[index1] := temp
        end
    end;

    procedure ChangeScores (var ID, Score: arraytype;
                            NumStudents: integer);
        var
            NeedToChange: char;
            found: boolean;
            position, OneID, NewScore: integer;
    procedure FindData (var a: arraytype;
                        count, value: integer;
                    var found: boolean;
                    var position: integer);
        var
            low, high, middle: integer;
    begin
        low := 1;
        high := count;
        repeat
            middle := (low + high) div 2;
            if value < a[middle] then
                high := middle - 1
            else
                low := middle + 1
        until (value = a[middle]) or (low > high);
        if value = a[middle] then begin
            found := true;
            position := middle
        end
        else
            found := false
    end;
begin
    write ('Scores to change (y or n): ');
    readln (NeedToChange);
    while (NeedToChange in ['Y', 'y']) do begin
        write ('Student ID: ');
        readln (OneID);
        write ('New Score: ');
        readln (NewScore);
        FindData(ID, NumStudents, OneID, found, position);
        if found then
            Score[position] := NewScore
        else
            writeln ('That id is not on file.');
        write ('More changes (y or n): ');
        readln (NeedToChange)
    end
end;
```

```
procedure ComputeStats (var a: arraytype;
                                count: integer);

    function Average (var a: arraytype;
                            count: integer)
                        : real;
        var
            index, sum: integer;
    begin
        sum := 0;
            for index := 1 to count do
                sum := sum + a[index];
            Average := sum / count
        end;

        function Max (var a: arraytype;
                            count: integer)
                        : integer;
            var
                index, big: integer;
        begin
            big := a[1];
            for index := 2 to count do
                if big < a[index] then
                    big := a[index];
            Max := big
        end;

    begin
        writeln ('Average is ', Average(a, count):5:1);
        writeln ('Maximum is ', Max(a, count))
    end;
begin
    OrderData(ID, Score, NumStudents);
    ChangeScores(ID, Score, NumStudents);
    ComputeStats(Score, NumStudents)
end

procedure RewriteData (var a1, a2: arraytype;
                                count: integer);
    var
        index: integer;
begin
    writeln ('Data Values');
    for index := 1 to count do begin
        writeln (a1[index]:10, a2[index]);
        writeln (outfile, a1[index]:10, a2[index])
    end
end;

begin
    reset(infile);
    rewrite(outfile);
    ReadData(ID, Score, NumStudents);
    ProcessData(ID, Score, NumStudents);
    RewriteData(ID, Score, NumStudents)
end.
```