Mental Representations of Programs by Novices and Experts

Vikki Fix
Computer Science Department
University of South Dakota
Vermillion SD 57069, USA
phone: (605), 677-5388
email: vikkif@charlie.usd.edu

Susan Wiedenbeck
Computer Science Department
University of Nebraska
Lincoln, NE 68588, USA
phone: (402) 472-5006
email: susan@cse.unl.edu

Jean Scholtz
Computer Science Department
Portland State University
Portland, OR 97207, USA
phone: (503) 725-4103
email: jean@cs.pdx.edu

ABSTRACT

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.

KEYWORDS: program comprehension, mental representation of programs

INTRODUCTION

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 disposal and how it is characteristically organized is crucial to understanding mental representations and predicting performance on comprehension-related programming tasks. 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. We propose that an expert's mental representation exhibits five abstract characteristics, which are generally absent in novice representations:

1. It is hierarchical and multi-layered;
2. It contains explicit mappings between the 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.

ABSTRACT CHARACTERISTICS

Some past research on program comprehension [1, 8, 9] has focused on the classes of information that programmers extract during program comprehension. An interest in this 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. A general finding has been that better comprehenders are distinguished from poorer comprehenders by their mastery of functional information. In the research reported here we do not focus on the classes of information making up the content of the representation but rather on the representation's general features. 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. While some of these features have been studied empirically, many of them have not been studied or have not been studied specifically in the context of program comprehension tasks. Each of the 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 [6]. A hierarchical structure of mental representations by programmers has been suggested by some observations of programmers carrying out programming tasks. Both Jeffries [4] and Nanja and Cook [7] 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, Boehm-Davis, and Schultz [3], 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, when comprehending a well-written program with clear hierarchical relations embedded in it.

An important feature of the mental representation is not just the existence of multiple layers of representation but the existence of **explicit mappings** between the layers. Letovsky [6] has argued that the overall goals of a program are usually readily understandable to programmers from such sources as mnemonic names 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 comprehending programs is to create a mapping between high-level goals and their code representation. In empirical work, Pennington [8] 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. According to Soloway and Ehrlich [10], programming knowledge is represented as a set of frame-like structures, called plans, for handling stereotypical situations which arise frequently in programming. 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 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. 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, i.e., plans in which the code implementing them is scattered throughout the program, rather than localized in one place [11]. 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 [5] 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 of where structures and operations occur physically 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 [4] observed that expert programmers, but not novices, were very skillful at locating information which they had previously seen in a program when it was needed a second time. 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.

METHODOLOGY

Subjects

Twenty novice and 20 expert programmers took part in the study. All of the subjects were volunteers. The novice programmers were undergraduate students who had recently completed a first semester Pascal course. The expert subjects were professional programmers with a median of 7 years of experience.

Materials

The program used in this study was written in Pascal. It was 135 lines long and occupied 3 printed pages. The program manipulated student data. The program consisted of a main program and 9 subprograms. The deepest nesting of program blocks was 4 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. Mnemonic identifiers were used in the program, which was also 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. Comprehension questions have been used as a method of measuring understanding in past research on program cognition (see[2]). The questions focused on determining whether programmers exhibited the abstract characteristics in their representations of the program. The questions required recalling information

Question Number	Question Content	Abstract Category	Significant Difference
1	Match procedure names to the procedures they call	Hierarchical structure	p=.0043
2	List procedure names	Hierarchical structure	n.s.
3	Write description of goals of selected procedures	Explicit Mappings	p=.0001
4	Write description of principal goals of program	Explicit Mappings	n.s.
5	Label complex code segments with plan label	Recurring patterns	p=.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=.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=.0007
10	Match variable names to the procedures in which they occur	Well grounded	p=.0006
11	Indicate physical location of invariant program parts	Well grounded	n.s.

Table 1: Summary of comprehension questions

about different objects or relations in the program The correspondence of the questions to the abstract characteristics is discussed below. The different questions relevant to one particular abstract characteristic were balanced to require a similar amount of information from the subject.

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. 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 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.

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 of two procedures in the program, telling what program goals they 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.

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. Question 6 also contained stereotyped patterns but ones of the most elementary kind, e.g., incrementing a counter. 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.

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 the major variables in the program. Generating these would indicate knowledge of the essential objects in the program, but it would not indicate well connectedness in the representation.

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 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: last 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 2 or 3. The subject was given a listing of the program to study for 15 minutes. They were told to study the program in detail in order to understand its structure and function as fully as possible. 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 an earlier question did not give away the answer to a later question. Subjects were not allowed to return to a previous questions once they had turned the page. They were allowed to take as long as they needed to work through the question booklet. For experts the times ranged from 40 minutes to 1 1/4 hours with the average around one hour. For novices the times ranged from 45 minutes to 1 1/2 hours with the average around 1 1/4 hours.

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 ($F(11, 28) = 8.9635$, $p = .0001$). 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 .0045, i.e, .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 ($F(1,38) = 9.20$, $p = .0043$). The expert 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 analyzed for the presence of 7 specific information elements. Experts scored significantly higher than novices on Question 3 ($F(1, 38) = 11.87$, $p = .0014$). The expert 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 analyzed subjects' statements about the methods which the subprograms used to achieve their goals. For this analysis, subjects' descriptions were scored for the presence of 4 specific information elements related to method. Experts also scored significantly higher on this measure ($F(1, 38) = 34.45$, $p = .0001$). The experts' mean was 2.90 out of the 4 elements, and the novices' mean was .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 ($F(1, 38) = 57.58$, $p = .0001$). 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 ($F(1, 38) = 11.99$, $p = .0013$). 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 7 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 ($F(1, 38) = 13.58$, $p = .0007$). The expert 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 ($F(1, 38) = 14.08$, $p = .0006$). 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 AND CONCLUSIONS

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. Experts extract many different kinds of information from a program which become a part of their mental representations. They are not distinguished from a novice along a single dimension or just a couple of dimensions. 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 quite significantly 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.

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 modules which are represented in the passage of data among modules. It may be that novice programmers do not pick up some information because they are using a different program comprehension strategy than experts, as suggested by Jeffries's observations [4] of a different order of program reading among novices. A different reading or study strategy may obscure some information selected by

experts and at the same time may highlight other information less useful to support programming tasks. We also suggest that novices may lack basic skills necessary to developing an expert-like representation, particularly skill at performing symbolic execution. Novices inability to carry out symbolic execution or their use of it in inappropriate circumstances has been noted in the past [4, 5]. Attention to study strategy and to appropriate use of symbolic execution in instruction may aid novices in developing more expert-like representations.

REFERENCES

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

2. Boehm-Davis, D. A. Software comprehension. In *Handbook of Human-Computer Interaction*, M. Helander, Ed., NY: North-Holland, 1988, pp. 107-121.

3. Holt, R. W., Boehm-Davis, D. A., and Schultz, A. C. Mental representations of programs for student and professional programmers. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 1987, pp. 33-46.

4. Jeffries, R. A comparison of the debugging behavior of novice and expert programmers. Paper presented at the American Educational Research Association Annual meeting, 1982.

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

6. Letovsky, S. Cognitive processes in program comprehension. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds., Norwood, NJ: Ablex, 1986, pp. 58-79.

7. Nanja, M. and Cook, C. R. An analysis of the on-line debugging process. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 1987, pp. 172-184.

8. Pennington, N. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds., Norwood, NJ: Ablex, 1987, pp. 100-113.

9. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19 (1987), 295-341.

10. Soloway, E. and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 10 (1984), 595-609.

11. Soloway, E., Pinto, J. Letovsky, S., Littman, D., and Lampert, R. Designing documentation to compensate for delocalized plans. *Communications of the ACM* 31 (1988), 1257-1267.