AN EMPIRICAL STUDY OF NOVICE PROGRAM COMPREHENSION IN THE IMPERATIVE AND OBJECT-ORIENTED STYLES

Vennila Ramalingam and Susan Wiedenbeck
Computer Science and Engineering Department
University of Nebraska
Lincoln, NE 68588-0115 USA
susan@cse.unl.edu

KEYWORDS: object-oriented programming, imperative programming, novices, mental representation, program model, domain model, program comprehension

ABSTRACT
The objective of this study was to determine whether the mental representation of object-oriented programs differs from imperative programs for novice programmers. In our study novices who had little or no previous programming experience studied and answered questions about three imperative and three object-oriented programs. The questions targeted information categories making up the program model and the domain model representations of the programs. It was found that there was a sharp contrast between the mental representations of the imperative and object-oriented programs. While the comprehension of the imperative programs was better overall than that of the object-oriented programs, the mental representations of the imperative programs focused on program-level knowledge. On the other hand, the mental representations of the object-oriented programs focused more strongly on domain-level knowledge. The results tend to support the view that language notations differ in how well they support the extraction of various kinds of information.

1. INTRODUCTION
Research on programming in the OO style has begun to appear, for example, the 1995 Special Issue of *Human-Computer Interaction* on object-oriented programming. However, the number and type of empirical studies of object-oriented programming are still fairly limited. To date, it appears that studies of OO programming have concentrated primarily on program design, and secondarily on reuse and maintenance. We are aware of only one study which focuses directly on comprehension in the OO style (Burkhardt, Détienne, and Wiedenbeck, 1997). Furthermore, most previous studies involve either OO experts or else experienced programmers learning to program in the OO style after substantial student or professional experience in the imperative style. Also, there have been few attempts to compare comprehension of programs written in the object-oriented and the imperative styles.

In this research, we investigated the comprehension by novices of small programs written in the imperative or the OO style. Our objective was to evaluate the mental representations formed during comprehension of the programs and, in particular, to compare the mental representations and comprehension of the imperative and the object-oriented programs. Thus, this work makes use of the comprehension model of Pennington (1987a, 1987b), which investigated the detailed mental representations formed by programmers studying programs written in the imperative style. Our question was whether we would find differences in the mental representations of the object-oriented and imperative programs which might be explained by characteristics of the respective paradigms.

The following section of the paper reviews related studies on program comprehension. The third section presents our methodology, the fourth section the results, and the fifth section a discussion of the results. Finally, the sixth section concludes with remarks about the limitations of this research and plans for future study.

2. BACKGROUND AND PRIOR RESEARCH

Program comprehension is the process of understanding a program written by oneself or someone else, normally for the purpose of doing some further task with the program which requires understanding. Program comprehension is a critical task in organizations for two reasons: 1) programmers change jobs frequently and new people are constantly being added to projects, requiring them to understand program parts that have already been written, and 2) most programming does not involve writing a program from scratch but instead starts from the basis of existing programs. In fact, a whole job category in the programming industry is "maintenance programmer," i.e., programmers who specialize in adding to or modifying the functionality of programs in use. These programming activities are founded on program comprehension. Other activities are also comprehension-related, e.g., program debugging, in which finding a bug often requires comprehending the buggy program, and code reuse, which requires comprehension of reusable components that will be incorporated into a new design.

Pennington's model of program comprehension (1987a, 1987b) is derived from influential models of text comprehension which have been developed and refined over the past 20 years (Johnson-Laird, 1983; Schmalhofer and Glavanov, 1986; van Dijk and Kintsch, 1983). These models of text comprehension are layered in that the reader is seen as forming a mental representation of a text which has multiple levels. This layered mental representation is then the basis for carrying out comprehension-demanding tasks with the text, for example, answering questions, summarizing, or paraphrasing it. The lowest layer in the mental model is the surface form representation, which is the reader's verbatim memory of the text. This representation is identical to the text or mirrors it closely. The textbase representation is more abstract than the verbatim representation. It contains knowledge about the propositions present in the text and the structure that propositions form in the text. This knowledge, known as text microstructure and macrostructure, is an abstraction from the verbatim surface of the text, but it is still based on propositions and relationships which are explicitly available in the text. A representation of the text microstructure and macrostructure is built automatically during reading. The highest level of abstraction is the situation model, which is the reader's representation of the situation in the world which the text describes. The situation model is based not just on information in the text, but rather on information in the text combined with the reader's own knowledge about the domain described in the text. The situation model is, thus, derived from inferences about the text which rely on reader's relevant knowledge. This being the case, it is clear that the formation and richness of the situation model depend strongly on the extent of the reader's domain knowledge and the effort made by the reader to draw inferences from the text (Mills, Diehl, Birkmire, and Mou, 1995).

Pennington's model of program comprehension (1987a, 1987b) applies the layered model of text comprehension to program texts. Beyond the surface form representation of the program text, Pennington describes two levels, the *program model* and the *domain model*. The program model is analogous to the textbase in text comprehension theory, while the domain model is analogous to the situation model. The program model is made up of operations and control flow knowledge. These two kinds of knowledge have in common that they are at a low level of abstraction, and they represent knowledge that is explicitly available in the program text. Operations knowledge has to do with specific elementary operations that are carried out in the source code. These operations are usually represented by one line of code, for example, incrementing a counter would be an elementary operation. Control flow knowledge has to do with the order in which lines of the source code are executed. The default order is sequential, but that may be modified by looping, branching, and calls to functions. Pennington's model proposes that programmers extract control flow information from a program text while reading and comprehending it. The domain model consists of data flow and function knowledge. Data flow concerns transformations which occur to variables as a program executes. These transformations change the data from its input state to its output state, so they are fundamental to the goals of a program. Because of this close connection to program goals, data flow knowledge may be considered a part of the domain model. Data flow is often difficult to understand in a program when variables form part of delocalized plans (Littman, Pinto, Letovsky, and Soloway, 1986), in which related data transformations are carried out in non-contiguous segments of the code. Function knowledge concerns the goals which a program accomplishes. For example, the overall function of a program might be to create a list of all students who are eligible to graduate with high honors. Program function, like data flow, can be difficult to determine because it is embodied, not in a single line of code or a few lines of code, but in an ensemble of coordinated program actions, or plans (Soloway and Ehrlich, 1984), perhaps spread across multiple program units. To understand the function of a program, the reader must invest the effort to gain mastery of these elements. Furthermore, understanding of the program's function may depend on the reader's pre-existing knowledge of the domain of the program. State knowledge was a fifth knowledge category included in Pennington's model. State relations are defined as "the connections between execution of an action and the state of all aspects of the program that are necessarily true at that point in time" (Pennington, 1987a, p. 101). Pennington (1987b) does not assign state knowledge to belong to either the program or the domain model, but observes that this type of knowledge would not likely be facilitated by the notations of existing programming languages.

Pennington carried out two empirical studies to test her model of program comprehension. In the first study (Pennington 1987a, 1987b), professional programmers studied brief programs written in an imperative language, either FORTRAN or COBOL according to the subject's expertise. After studying each program, the subject answered from memory a set of five questions, one each on operations, control flow, data flow, function, and state. It was found that a strong program model was formed, i.e., the mental representation was dominated by information about operations and control flow. Domain model information was more poorly represented; and, in particular, function knowledge was often lacking. However, the more expert programmers (i.e., those who scored highest in terms of total score across the question sets) showed a tendency to have a higher proportion of correct function information available in their representations. State information was poorly represented in the mental representations of all subjects.

In a second experiment (1987a, 1987b) professional programmers first studied a longer imperative program, then answered questions about it in the same five information categories. In a second phase of this experiment, they carried out a program modification, then answered another set of comprehension questions. The results of the first question set, given after the study phase, showed a similar pattern of results to the short programs: the program model dominated and there was a much higher error rate on the domain model. After the program modification, the results changed. The domain model became dominant with low error rates on function and data flow knowledge. Error rates did not decrease, or even increased a bit, on program model knowledge. Pennington uses the results of these two studies to argue for the cognitive validity of the distinction between the program model and the domain model. She also uses the results of the modification phase in the second experiment to argue that the performance of comprehension-demanding tasks is likely to play an important role in the formation of the domain model.

In a related study, Berganz and Hassell (1991) collected think-aloud protocols of PROLOG experts comprehending a small PROLOG program. From the protocols they extracted the number of verbalizations falling into different knowledge categories and the temporal ordering of verbalizations about the knowledge categories. Their results supported Pennington's dual model. PROLOG experts concentrated first on data structures, which form a part of the program model, and later on function, which is the key element of the domain model. Interestingly, data flow relationships did not appear to be important in the comprehension of the PROLOG program.

Corritore and Wiedenbeck (1991) applied Pennington's model to the study of program comprehension by novice imperative programmers. Subjects who were at the beginning of a second programming course studied and answered questions about small Pascal program segments. The questions fell into the five categories used by Pennington. It was found that novice programmers formed mental representations which were strongly program oriented, with little domain-level representation. The correctness of their responses was highest on operations questions, which were very concrete and closest to the surface representation of the program text. In a second experiment, the same novice subjects studied a longer Pascal program and answered questions about it. The results showed that a program-level representation dominated. The domain model was not well developed, and there was an exaggerated rate of errors on function questions.

The current work is also related to the stream of research on information extraction in programming. This work concerns programming language notations. The gist of it is that there is no superlative notation that is best for all tasks and all users. Instead, a given notation may facilitate or fail to facilitate a specific programming task. It usually is the case that a notation which favors a certain task penalizes some other task. For example, Gilmore and Green (1984) compared procedural and declarative notations on a question answering task. They found that the procedural notation was superior for answering sequential questions, i.e. questions about what happens in a program after some action X is performed. On the other hand, the declarative notation was better for answering circumstantial questions, i.e. questions about what combination of circumstances in a program will cause action X to be performed. This phenomenon of information accessibility in programs was referred to by Gilmore and Green as the 'match-mismatch conjecture.' In a similar fashion, work comparing textual and graphical notations (Green, Petre, and Bellamy, 1991; Moher, Mak, Blumenthal, and Leventhal, 1993) has shown that graphical notations are not an ultimate panacea, in spite of claims of them being easier to read, giving an overview of program structure, and supplying more information. Instead, the critical questions about a graphical notation, as about any notation, are to what extent does the notation make certain types of information accessible, and does the user have adequate experience with the notation type. Good (1996) compared a control flow and data flow representation for presenting PROLOG programs. Contrary to the previous research, she failed to find that the two representations facilitated answering questions about the corresponding information categories. However, a more detailed analysis of the representations and the questions suggested that, even within a type such as data flow, the structure of the representation may be such that it does not support well all data flow questions. Thus, the idea of a match between a task and a notation needs to be expanded to include the idea of degrees of matching.

The research on information extraction is highly relevant to this study because superlativist claims have been made about object-oriented languages. Advocates have argued that the OO style of programming is more "natural" (Borgida, Greenspan, and Mylopoulos, 1986; Rosson and Alpert, 1990). The argument is that OO design, with its focus on active objects, their relationships, their behaviors, and their interactions, provides a better match to the way that designers conceptualize problems than does the alternative of decomposition into procedures which act on passive data structures (Rosson and Alpert, 1990). In effect, designers in the OO style are seen as working at a level that is closer to the domain of the problem they are solving (Booch, 1986; Borgida, Greenspan, and Mylopoulos, 1986). Rosson and Alpert (1990) further suggest that, if the OO style is indeed more intimately connected to the problem domain, then there should be benefits not just in program design but also in program maintenance, comprehension, and reuse.

Given the strong empirical evidence against superlativist claims, we approach the comparison of comprehension of imperative and object-oriented style programs from the more sustainable viewpoints of mental model formation and information accessibility. One research question is the nature of novices' mental representation of programs in the two styles. A second research question is whether there are differences in information extraction between the two styles which might be explained by characteristics of the styles and the task performed.

3. METHODOLOGY

In this experiment novice programmers studied six brief C++ program segments. Three were written in the imperative style and three were written in the object-oriented style. After studying a segment, the program was hidden. The subjects immediately answered from memory a set of five comprehension questions about it, one each in the categories: operations, control flow, data flow, state, and function. The analysis is based on the errors in subjects' responses.

3.1. Subjects

Seventy-five subjects participated in the study. The subjects were students at a large university and were enrolled in an introduction to programming course using C++. The course was divided into a number of small sections taught by different teachers, and students from four sections participated. The sections of the course were closely coordinated by a lead teacher, who monitored that the sections covered the same material at approximately the same pace. A common textbook was used across all sections of the course, and common programming assignments were given. Participation in the study was voluntary. Students who participated were given a small amount of credit toward their final grade in the course.

A background questionnaire, which focused on subjects' computer and programming experience, was administered before the experimental task. There were 25 female and 50 male subjects. The subjects came from a variety of undergraduate majors, and slightly less than half were computer science majors. Most of the subjects were second or third year undergraduate students. Their average age was 20 years. On average the subjects had first been introduced to computer use in junior high school. They had taken an average of 1.27 programming courses in high school and college (exclusive of the course from which they were recruited), and they were familiar with 1.37 programming languages (including C++, which they were learning in the current course). They reported having taken 2.75 courses in high school or college in which they used computers as tools, for example word processing, spreadsheets, or educational software. The mean number of programs they had written was between 11 and 20. On average they estimated their longest program as 21-50 lines of code.

The course in which the subjects were enrolled taught problem solving and programming in C++. It began with the fundamentals of programming, including basic data types, variables, assignment, arithmetic operations, comparisons, branching, looping, and functions. After several weeks on the fundamentals, object-oriented concepts were introduced. The object-oriented concepts taught at this point included classes, encapsulation, overloading, and message passing. Once these object-oriented concepts were motivated and discussed, students were presented with many examples of programs written in the object-oriented style in both their lectures and textbook. Students then began writing programs of their own using classes, encapsulation, and message passing. More advanced object-oriented concepts, such as inheritance and polymorphism, were also discussed in order to further motivate the OO style. However, while students were aware of these features, they did not see examples or write programs of their own using them.

3.2. Materials

The materials consisted of six programs. All of the programs were written in C++. In order to be able to compare subjects' comprehension in the two styles, three of them were written using object-oriented features of C++, while the other three were written without object-oriented features. This was accomplished by taking advantage of the fact that C++ is an object-oriented superset which contains imperative C, so a student learning C++ essentially learns C as well. For convenience we refer to the two program styles as object-oriented and imperative, while acknowledging that C and C++ are both imperative languages at base.

The object-oriented programs each contained a class, in which the main computation of the program was done. The main procedure instantiated one or more objects of the class and then passed a message to a function of the class asking it to do a computation. The OO style programs were brief, their listings fitting on a single sheet of paper. Each OO style program contained only one class and did not use more complex OO features, such as inheritance and polymorphism. These features were not incorporated because they were beyond the level of programming experience of the subjects. The programs were also designed with the intention of excluding the effect of domain knowledge from the subjects' performance. This was done by selecting domains that were within the normal experience of all subjects. Program A declared a class Rectangle and contained a function which determined whether a given Rectangle object was a square. Program B defined a Car class and contained a function which could determine whether a car object was traveling above the speed limit. Program C declared a Washing_Machine class and contained a function to simulate the actions of a washing machine.

The imperative style programs were written in the imperative C subset of C++, with the one exception that they used C++ stream I/O, since the subjects were familiar with those conventions. The programs did not contain objects, classes, or message passing. Instead they contained simply a main program. The main program declared variables and then carried out the computations of the program. The imperative programs were slightly shorter than the object-oriented programs since the OO programs had the overhead of class definitions. The three imperative programs were used previously in a study by Corritore and Wiedenbeck (1991), and were used here for comparison with the earlier study. In this experiment the questions associated with the programs were slightly different. Program D read in two numbers, summed the pair of numbers, and printed out the sum with identifying messages. These actions were embedded in a loop which repeated the activities 10 times. Program E read in an amount of change from 1 to 99 cents, then calculated how to make change in the largest denominations possible (quarters, dimes, etc.). Program F read in integers until a sentinel value. A running sum and count were kept, and at the end of the input the average was calculated and printed. Like the OO programs, all three programs could be understood without specialized domain knowledge on the part of the subjects.

Five comprehension questions were created for each program. There was one question each on operations, control flow, data flow, state, and function. Operations questions asked about some single line operation in the program, such as whether a certain variable was assigned a particular value. Control flow questions asked about the order in which operations were carried out, i.e., whether a certain operation was executed before some other operation. Data flow questions asked about data transformations. As in Pennington's work (1987a, 1987b) the questions were stated to ask whether a certain variable affected the value of another variable in the program. State questions asked the state of a certain variable when a certain point in the program was reached. Finally, function questions asked about the goals achieved in the program, i.e., what the program did. Each question had a yes/no answer. Appendix A shows sample programs and questions for the imperative and object-oriented styles.

3.3. Procedure

Subjects were run in a class session during the twelfth week of a fifteen week semester. At this point subjects had studied basic language structures through functions, classes, and arrays. They had written programs targeting all of the features used in the experimental programs The testing was done with paper and pencil. Subjects first filled in a demographic questionnaire. Then they were given a booklet containing the programs and their associated question sets. Each program was printed on a single page by itself. The questions for the program followed on the next page. The order of presentation of the programs was counterbalanced. Subjects were given 2 minutes to study the program, and were instructed not to turn the page before being told to do so by the experimenter. After that, they were given 1.25 minutes to answer the questions. These times were chosen after pre-testing was done to determine how much time subjects needed to study the programs and answer the questions. The same procedure was repeated for each of the six programs.

3.4. Limitations of the Design

Several limitations of the experimental design should be borne in mind in reading and interpreting the results.

Longer exposure to the C subset. The subjects had longer exposure to the C subset than to C++. They had used the imperative C constructs, which formed the basis for the imperative, programs, for 12 weeks. The C++ concepts of classes, encapsulation, and message passing had been used for about 8 weeks.

Lack of a criterion for comparability between programs. Because the experiment used a repeated measures design, different problem sets were used for the imperative and OO cases. While these were all brief and simple programs, we have no criterion of comparability to use to argue about their "equivalence" for experimental purposes. Notably, the OO programs were systematically slightly longer than the imperative programs because of the overhead of declaring classes. The program sets also differed in that the OO programs carried out their main computation in a member function of a class, while the imperative programs carried their main computation out in the main program.

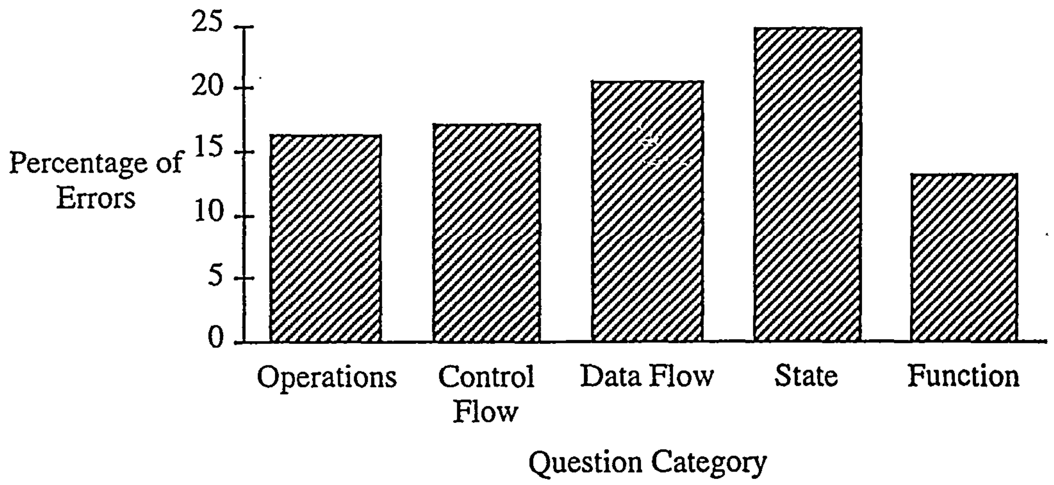
Representativeness of the materials. The programs were very small. Certain features of C were not used, including function calls and parameter passing. Certain defining features of the OO style were also not represented, including inheritance and polymorphism. This limits the representativeness of these results.

4. RESULTS

A preliminary analysis was done to determine whether there was a significant difference in performance among subjects enrolled in the four different course sections. A one-way ANOVA was run with section as the independent variable and number of errors on the comprehension questions as the dependent variable. The result was not significant. Therefore, we did not include section as a variable in further analyses.

Our data analysis began with the whole set of six programs. The mean percentage of errors across all questions on the six programs was 18.36. The mean percentage of errors in each question category is shown graphically in Figure 1. A one-way repeated measures Analysis of Variance was used. The independent variable was the question category with five levels: operations, control flow, data flow, state, and function. The dependent variable was percentage of erroneous responses out of the total number of questions in each question category. The ANOVA was significant (F(4, 384) = 10.96, $p < .0001$). Newman-Keul's test was run as a follow-up. It showed that there were significantly more errors on state questions than on operations, control flow, and function questions ($p < .05$). The only other significant difference was that there were more errors on data flow than on function questions ($p < .05$).

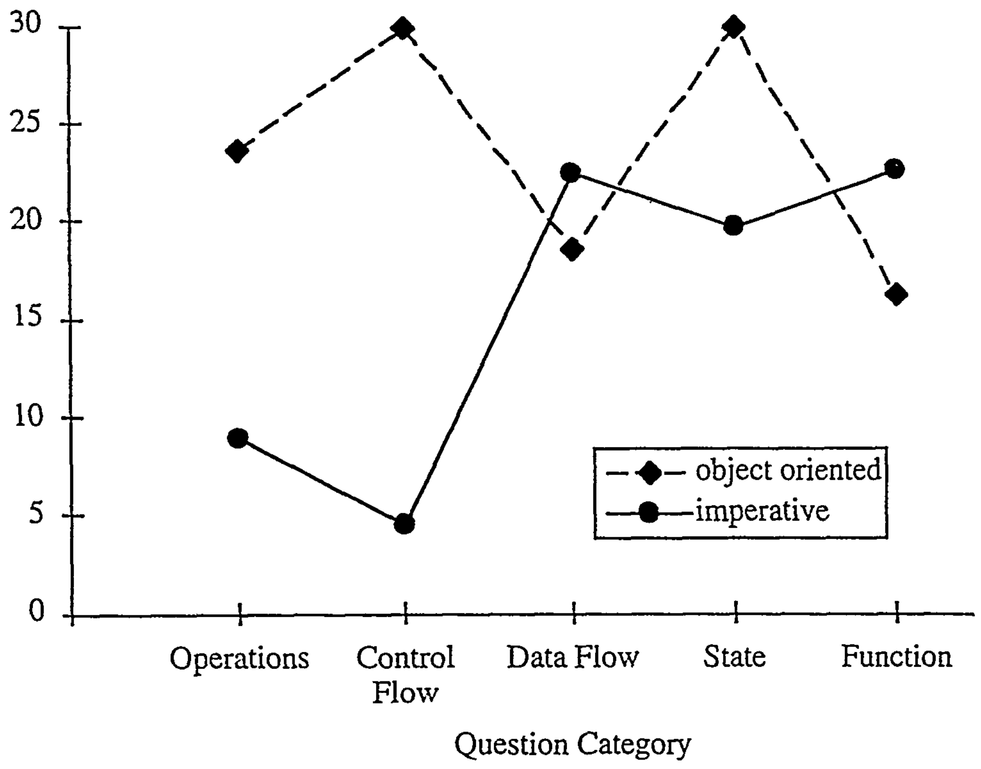
Figure 1. Mean error percentages in the five question categories for all programs



The one-way analysis represents overall performance on the comprehension questions, but it does not account for possible differences based on program style. Thus, another analysis was done to investigate the influence of the two program styles. The mean error percentage of the imperative group across all question categories was 13.05 (sd = 10.49). For the object-oriented group, the mean error percentage across all question categories was 23.64 (sd = 12.73).

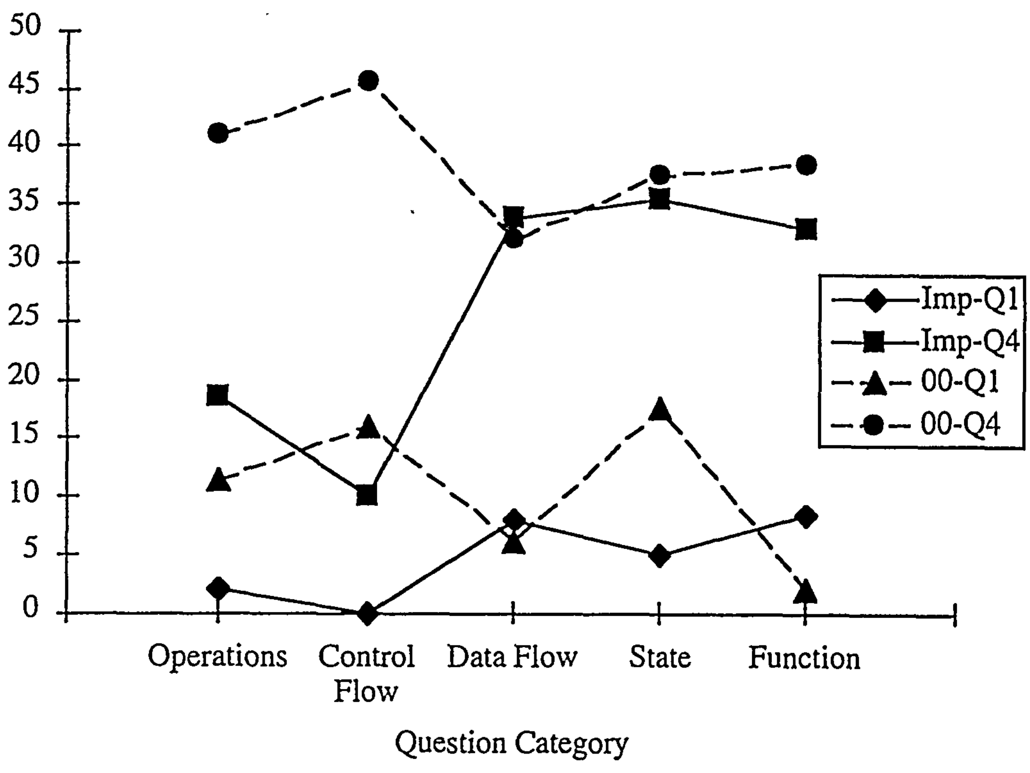
The mean error percentages for the imperative and object-oriented program sets broken down by the five question categories are graphed in Figure 2. A two-way within subjects Analysis of Variance was carried out. One independent variables was question category, consisting of five levels: operations, control flow, data flow, state, and function. The other independent variable was program style: imperative or object oriented. The dependent variable was percentage of erroneous responses out of the total number of questions in each question category. The ANOVA showed that there was a significant main effect of question category (F(4, 384) = 10.96, $p < .0001$) and of style (F(1, 96) = 56.22, $p < .0001$). There was also a significant two-way interaction of question category and style (F(4, 384) = 13.92, $p < .0001$). Newman-Keul's test was used for follow-up testing. It showed that there were differences both within and between the two programming styles. Within the imperative style, errors on operations and control flow questions did not differ from each other, but they were significantly lower than the errors on data flow, state, and function questions ($p < .05$). Within the object-oriented style, data flow and function questions had significantly lower errors than control flow and state questions. ($p < .05$), while operations questions did not differ significantly from any of the other categories. Between styles there were significant differences on operations, control flow, state, and function ($p < .05$).

Figure 2. Mean errors percentages in the five question categories for the imperative and object-oriented program sets



Finally, we compared the subjects with the lowest overall error rate across all question categories, Quartile 1, to those with the highest overall error rate, Quartile 4. The objective was to see whether the more skilled subjects differed from the less skilled subjects in their mental representations of the imperative and object-oriented programs. The middle two quartiles were dropped out of this analysis. The means are shown in Figure 3. A three-way mixed model Analysis of Variance was run. The between subjects factor was quartile with two levels: Q1 (n = 21) and Q4 (n = 22). The within subjects factors were question category with five levels and style with two levels, as in the previous analysis. The quartile factor was significant (F(1, 41) = 332.39, p < .0001). Question category approached significance (F(4, 164) = 2.02, p < .094). Style was not significant. Of the two-way interactions, only the question category by style interaction was significant (F(4, 164) = 5.02, p < .001). The three-way interaction was not significant. Follow-up tests were not performed because the two and three-way interactions involving quartile, the new factor in this analysis, were not significant.

Figure 3. Mean error percentages in the five question categories for Q1 and Q4 in the imperative and object-oriented program sets



5. DISCUSSION

The comparison of the imperative programs to the object-oriented programs showed that there were higher overall error rates in the OO style. We cannot explain this definitively, but have previously suggested some methodological issues which may have played a role, particularly the question of the comparability of the C and C++ problem sets and the longer time the subjects had spent working with the C subset of C++. The comparability issue could be addressed in a future study by using a fully randomized design employing the same programs written in the two different styles. On the other hand, if the explanation of the overall difference between styles lies in lesser familiarity with OO features and what they entail, we would expect the gap between the OO and imperative programs to diminish with further training in the OO paradigm. An interesting question is how long the gap would persist and if the learning curve is steeper for C++ than for C.

While, the question of why the imperative style had fewer overall errors is interesting, the main point of our comparison is not the overall errors but the nature of the mental representation formed in the two programming styles. This comparison of mental representations may still be made, and be very informative, in spite of the difference in overall error rates. Considering the performance in the different question categories on the imperative and the OO programs, we find clear evidence of differences in mental representation between the two styles. As Figure 2 shows, the pattern of response to questions on the two program sets was very distinct. For the imperative style, there were few errors on operations and control flow questions, but much higher errors on data flow, function, and state questions. This supports the argument that operations and control flow information was easily accessible in the programs, and that subjects formed a *program-level* mental representation. This program-level representation is comparable to the textbase representation in text understanding, i.e., a representation based on relationships explicitly given in the text. Generally, the results are quite similar to those found by Pennington (1987a, 1987b), Berganz and Hassell (1991), and Corritore and Wiedenbeck (1991) in their studies of mental representation in other procedural languages (FORTRAN, COBOL and Pascal). In these prior studies, the program model was dominant in the mental representation after initial study of a program, as it was here. Thus, the imperative programs in this study, using the C subset of C++, provide further support for Pennington's model with respect to imperative programming: the program model and domain model appear to be distinguishable entities containing operations and control flow information on the program side vs. data flow and function information on the domain side. The state category, which Pennington did not assign to either the program or domain model, had a high error rate here, as in prior research. This again supports Pennington's contention that state information is difficult to extract from programs and not very available in programmers' mental representations of imperative programs.

For the OO style programs, the error rate was higher for control flow questions than for data flow and function questions. State questions had high error rates as in the imperative program set and in all past research. These results suggest that data flow and function information was accessible in the program and that subjects used it to form primarily a *domain-level* mental representation. It is notable that, in spite of higher overall error rate on the OO program set, the error rates on data flow and function questions were less than for the imperative program set. The program model, on the other hand, was weaker for the OO program set than for the imperative program set. The finding of a strong domain-based representation after initial study of an OO program contrasts with Pennington's and Corritore and Wiedenbeck's findings of a strong program model after initial study of an imperative program. However, in one of her studies of imperative programmers, Pennington (1987a) also measured the mental representation of her subjects in a second time frame, after the subjects had performed a modification of the program. Here the results were strikingly similar to our results for the OO programs after initial study: errors were low on domain model questions but high on program model questions. Pennington argues that this indicates that the domain model is built more slowly through working with a program in the context of a meaningful programming task.

The fact that a strong domain representation was built in *initial* study of the program in this experiment may be interpreted as support for Gilmore and Green's results (1984), showing that extracting information from programs is easier when the language structure reveals the kind of information in question and harder when it does not. Brooks (1983) portrays program comprehension as reconstructing all or part of the mappings which were made by the original programmer between a problem and the programming artifact. This experiment suggests that the OO style, as implemented in these simple programs, made some of the mappings more salient, namely the critical mappings of data flow and function.

Pennington reported that the subjects who comprehended best were those with "cross referenced" mental representations, i.e., those who developed a mental model containing a fairly balanced mix of domain and program-level representation. Subjects who built a strong program model with little domain representation could fail to grasp the ultimate goals of a program. On the other hand, subjects who built a strong domain model with little program representation could end up with an understanding of a program which was too high-level and disembodied from the code to support programming tasks. The ideal combination is a balanced representation. With respect to the current results, it is difficult to say whether the strong domain model representation of the OO style programs and the weaker program model representation is balanced enough to effectively support programming tasks. However, it may be observed (Figure 2) that the contrast in scores between the program model and the domain model was less in the OO style than in the imperative style. This suggests that there may be greater balance between the program and domain models in the OO style. However, further research is needed on this question.

The breakdown of the results by quartile (Figure 3) shows that in Q1, the better comprehenders, the pattern of results on both the imperative and OO programs is very similar in terms of the general shape of the curve to that of all subjects (Figure 2). In Q4, the poorest comprehenders, the shape of the curve for the imperative programs is similar to that of all subjects. However, the relative distribution of errors on the OO programs in Q4 looks slightly different, with an error rate on state questions that is proportionally less extreme, combined with a proportionally *higher* rate of errors on function questions. The most interesting trend in the quartiles data is that the rate of errors on function questions is extremely low on the OO style for Q1 ($m_{Q1\text{-function}} = 1.59$ percent), while the error rate on these questions in Q4 does not drop off as in Q1($m_{Q4\text{-function}} =$ 39.39 percent), or as is seen in the data for all OO subjects in Figure 2. Thus, the best comprehending subjects built a mental representation that had a very strong component of function knowledge. Since these subjects were novices, we may assume that the subjects in Q1 were more advanced on the learning curve of C++. It may be that the poorer comprehenders had more difficulty assimilating and taking advantage of the object-oriented nature of the OO programs. Thus, a key component of their domain model was less developed.

6. CONCLUSION

The results of this study indicate that novice programmers comprehending an OO style program form a strong domain model, while novices comprehending an imperative style program form a strong program model. In the experiment this was shown by the low error rate in the imperative style on both operations and control flow questions, which together make up the program model. Correspondingly, in the OO style there was a low error rate on data flow and function questions, which make up the domain model. Thus, the two styles differed in the nature of the mental representation formed by subjects during study of a program.

We interpret the results in terms of building mappings. In program design the problem is establishing mappings between real world entities and their representation in a program. In program comprehension the problem is making reverse mappings from the given program to an understanding of the real world entities and actions involved. This research suggests that the OO style facilitates the mapping from the program to the domain for novice programmers working on small and simple programs. This may be because there is more explicit and salient domain-related information in the OO style programs than in the imperative style programs. On the other hand, the OO style programs appear to have obscured operations and control flow.

The present study is one of what should eventually be an ensemble of empirical studies of object-oriented program comprehension. While this study did find distinct differences in the mental representation of imperative and OO style programs, its limitations leave many questions which provide directions for future research. Also, the results raise new questions which may be pursued. These issues are discussed in the following paragraphs.

One important question which arises from our results concerns the overall level of comprehension of the OO programs. It was found that novices comprehending the OO style programs formed a stronger domain model than program model. However, at the same time, they performed more poorly on overall comprehension of the OO style programs then they did on the imperative style programs. There are several reasons why this might be the case, as suggested earlier. First, the overall comprehension difference might be mitigated if the experimental materials and procedure were redesigned. Second, the difference might be related to the learning curve of the OO style being higher. Here the argument would be that OO programmers must learn the basics of the imperative style, including data types, assignment, branching, looping, and functions, as well as the OO features. There may simply be more for novices to master in the OO style, resulting in a steeper learning curve. We believe that this is a plausible explanation of our results. We have some preliminary evidence that supports the steeper learning curve hypothesis and suggests that the learning curve problem continues beyond the first programming course (Wiedenbeck, Ramalingam, Sarasamma, and Corritore, 1997). It appears that longer term research which studies the learning curve of OO novices is called for. Only then will we be able to evaluate what the higher overall error rate in the OO style means.

Given our interpretation that this experiment supports the view of the match-mismatch conjecture, it is important not to over-interpret the results. Two specific computational models were contrasted, both embedded in the C/C++ language. This one experiment is not representative of all form of OO and imperative languages. Other OO languages, such as Smalltalk and Eiffel, may differ in significant ways from C++, and these differences may affect information extraction and consequent mental model building.

Along these same cautionary lines, the experiment does not tell us how the OO style affects comprehension and mental model building of expert programmers working on more complex programs. We have no indication of whether experts would form a strong domain model during comprehension of a large OO program. From a pedagogical viewpoint it is important to know what is the effect of the OO style on learners, which we have studied here. However, it is equally important, if not more so, to understand how the OO style affects professional programmers. The finding of differences in the mental representation of imperative and OO programs in this study makes further study of such questions interesting. Work on the mental representation of OO programs by experts is in progress (Burkhardt et al., 1997).

Another limitation is this experiment is that it failed to ask questions specifically about the static contents of classes, which was an essential difference between the OO and imperative styles, as implemented in our materials. Thus, for instance, we do not have direct evidence about whether the subjects gained a good mental representation of the attributes of objects. The reason for this omission was to make it possible to ask the same categories of questions about both the OO and imperative style programs, thus facilitating a comparison. Other OO features, such as inheritance and polymorphism, were not investigated because they were beyond the scope of the experience of our novice subjects. However, further studies of comprehension are needed which determine the role played in the mental representation of specifically object-oriented features. This questions of the role of objects and other OO features in the mental representation merits further study. Preliminary results from a study of OO experts (Burkhardt et al., 1997), suggest that classes and the relationship of classes form an important part of their mental representations. The role of other OO features in mental representations is also is of interest and may be studied for experts or advanced novices in more complex programs.

Pennington's work suggests that a cross referenced mental representation, containing a balanced mix of program and domain knowledge, is associated with better program comprehension. It appears that skilled program comprehenders consider the program world and the domain world simultaneously and explicitly link their understanding of the two. In our study, the mental representations of the novice subjects did not appear to have great balance. The imperative style was associated with a strong program model and the OO style with a strong domain model. Further study is required to determine whether, and at what stage of development, novices begin to develop more balanced mental representations, in either style.

Pennington found the emergence of a cross referenced mental representation after the performance of a modification task. Our experiment required study of the programs for comprehension but did not involve the performance of a further comprehension-demanding programming task. While a task of reading and answering questions about a program is not unusual for novices enrolled in computer science courses, Pennington's results suggest the need for further studies which incorporate a task element. It may also be necessary to study novices at different levels of development to observe the emergence of comprehension strategies. Furthermore, if performance of programming tasks changes the nature of programmers' mental representations, then it is also necessary to study how *different* tasks affect the representation. Different comprehension-demanding tasks, such as modification vs. reuse, may affect the mental representation in different ways. Even a single task type, such as modification, may have different effects on the mental representation depending on the kind of modification, for example a localized vs. delocalized modification (Littman et al., 1986; Koenemann and Robertson, 1991). Thus, not just task effect in a global sense, but the specific nature of the task, may be of considerable importance.

REFERENCES

Bergantz, D. and Hassell, J. (1991). Information relationships in PROLOG programs: how do programmers comprehend functionality? **International Journal of Man-Machine Studies**, 35, 313-328.

Booch, G. (1986). Object-oriented development. **IEEE Transactions on Software Engineering**, SE-12(2), 211-221.

Borgida, A., Greenspan, S., and Mylopoulos, J. (1986). Knowledge representation as the basis for requirements specifications. In C. Rich and C. R. Waters (Eds.), **Readings in Artificial Intelligence and Software Engineering** (pp. 561-570). Los Altos, CA: Kaufmann.

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

Burkhardt, J-M., Détienne, F., and Wiedenbeck, S. (1997). Mental representations constructed by experts and novices in object-oriented program comprehension. **INTERACT'97 Conference Proceedings**, to appear.

Corritore, C. L. and Wiedenbeck, S. (1991). What do novices learn during program comprehension? **International Journal of Human-Computer Interaction**, 3(2), 199-222.

Gilmore, D. J. and Green, T. R. G. (1984). Comprehension and recall of miniature programs. **International Journal of Man-Machine Studies**, 21, 31-48.

Green, T. R. G., Petre, M., and Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: a test of superlativism against the'match-mismatch' conjecture. In J. Koenemann-Belliveau, T. G. Moher, and S. P. Robertson (Eds.), **Empirical Studies of Programmers: Fourth Workshop** (pp. 121-146). Norwood, NJ: Ablex.

Good, J. (1996). The 'right' tool for the task: an investigation of external representations, program abstractions and task requirements. In W. D. Gray and D. A. Boehm-Davis (Eds.), **Empirical Studies of Programmers: Sixth Workshop** (pp. 77-98). Norwood, NJ: Ablex.

Johnson-Laird, P. N. (1983). **Mental models: Towards Cognitive Science of Language, Inference, and Consciousness.** Cambridge: Cambridge University Press.

Koenemann, J. and Robertson, S. (1991). Expert problem solving strategies for program comprehension. **CHI'91 Proceedings** (pp. 125-130), New York: ACM.

Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. (1986). Mental Models and Software Maintenance. In E. Soloway and S. Iyengar (Eds.), **Empirical Studies of Programmers** (pp. 80-98). Norwood, NJ: Ablex.

Mills, B. C., Diehl, V. A., Birkmire, D. P. & Mou, L-C. (1995). Reading procedural texts: effects of purpose for reading and predictions of reading comprehension models. **Discourse Processes**, 20, 79-107.

Moher, T. G., Mak, D. C., Blumenthal, B., and Leventhal, L. M. (1993). Comparing the comprehensibility of textual and graphical programs: the case for Petri nets. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer (Eds.), **Empirical Studies of Programmers: Fifth Workshop** (pp. 137-161). Norwood, NJ: Ablex.

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

Pennington, N. (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. **Cognitive Psychology**, 19, 295-341.

Rosson, M. B., and Alpert, S. R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction,* 5(4), 345-379.

Schmalhofer, F., and Glavanov, D. (1986). Three Components of Understanding a Programmer's Manual : Verbatim, Propositional, and Situational Representations. **Journal of Memory and Language**, 25, 295-313.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. **IEEE Transactions on Software Engineering**, SE-10(5), 595-609.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1982). What do novices know about programming? In A. Badre and B. Shneiderman (Eds.), **Directions in Human-Computer Interaction** (pp. 27-54). Norwood, NJ: Ablex.

van Dijk, T. A. and Kintsch, W. (1983). **Strategies of Discourse Comprehension.** New York: Academic.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. L. (1997). **A Comparison of the Comprehension of Object-Oriented and Procedural Programs by Novice Programmers.** Departmental Report, Computer Science and Engineering Department, University of Nebraska-Lincoln.

APPENDIX A

1. IMPERATIVE STYLE PROGRAM

```
#include <iostream.h>

int main()
{
        int Amount;
        cout << "\nEnter an amount of change from 1 to 99 cents \n";
        cin >> Amount;
        int Quarters = Amount / 25;
        int AmountLeft = Amount % 25;
        int Dimes = AmountLeft / 10;
        AmountLeft = AmountLeft % 10;
        int Nickels = AmountLeft / 5;
        AmountLeft = AmountLeft % 5;
        int Pennies = AmountLeft;
        cout << Amount << " can be given as : " << Quarters << "quarters " << Dimes <<
                "dimes " << Nickels << "nickels " << Pennies << "pennies.\n";
        return 0;
}
```

Questions

1. Is the variable Pennies initialized to 0?

....[Yes / No]

2. Is the number of quarters needed calculated before the number of dimes needed?

....[Yes / No]

3. Will the value of AmountLeft affect the value of Pennies?

....[Yes / No]

4. Does AmountLeft have a value before Quarters is assigned a value?

....[Yes / No]

5. Does this program compute how to give change in the largest possible denominations?

....[Yes / No]

2. OBJECT-ORIENTED STYLE PROGRAM

```
#include <iostream.h>
class Car
{
private:
        int Passengers, Speed;
public:
        Car(int p, int s);
        void check_speed_limit();
};

Car::Car(int, int)
{
Passengers = p;
if (p == 0)
        Speed = 0;
else
        Speed = s;
}

void Car::check_speed_limit()
{
if (Speed >= 55)
        cout << "Over the limit! Slow Down!!! \n";
}

int main()
{
Car mycar(1, 25);
mycar.check_speed_limit();
return 0;
}
```

Questions

1. Is the speed of mycar set to 25?

.... [Yes / No]

2. Is the output statement executed before the speed is checked?

....[Yes / No]

3. Does the value of Passengers affect the value of Speed?

....[Yes / No]

4. When the cout statement is reached, is the value of Speed less than 55?

....[Yes / No]

5. Does the program compare the speed of two cars?

....[Yes / No]