Navigation and Comprehension of Programs by Novice Programmers

Russell Mosemann
Computer Science Department
Concordia University
Seward, NE 68434 USA
+1 402 643 7445
mose@seward.cune.edu

Susan Wiedenbeck
College of IS&T
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104 USA
+1 215 895 2490
susan.wiedenbeck@drexel.edu

ABSTRACT
The purpose of this research is to examine the influence of different methods of program navigation on the mental representation and comprehension of novice procedural programmers. As a programmer tries to comprehend a program, a particular navigation method may assist or inhibit the process by highlighting, or making more accessible, certain kinds of information. Presumably, a method of navigation that highlights a certain type of information will help the programmer to better comprehend that information. In this research we study the effect of sequential, control flow, and data flow navigation methods on novices programmers' overall comprehension of a program and on the ability to comprehend specific types of information. Our results indicate that novice comprehension is facilitated by a sequential or control flow view of the program and is inhibited by a data flow view.

Keywords
Program comprehension, navigation, novice programmers, empirical studies

1 INTRODUCTION

A programmer comes into contact with the source code of a program on two occasions: during the design and construction of the program and after the program has been written. Common reasons for looking at the source code after the program has been written include debugging, modification, and code reuse. Professional programmers often read a program to understand how to structure changes they make and integrate them with the existing code. Novices learning a programming language frequently read sample programs to understand language concepts.

An initial step on the road to comprehending a program involves how the programmer works through, or navigates, the program text. Navigation includes not only the path the programmer takes in reading the program, but also the gathering of information along the way, the recognition of the current location in relation to the rest of the program, and the ability to find the way to a desired new location. Research on program comprehension has suggested differences between novice and experienced programmers in their spontaneous methods of navigating programs and in the resulting quality of their comprehension [8, 11]. These results, which suggest a preference for sequential program navigation among novices, are intriguing and have been widely cited. However, to our knowledge there are no published studies of programmers in which the method of program navigation is systematically manipulated in order to determine its effect on program comprehension.

The purpose of this research is to examine the influence of different methods of navigation on the mental representation and comprehension of novice procedural programmers. As a programmer tries to comprehend a program, a particular navigation method may assist or inhibit the process by highlighting, or making more accessible, different kinds of information. Presumably, a method of navigation that highlights a certain type of information will help the programmer to better comprehend that information [5]. In this research we study the effect of different navigation methods on novices programmers' overall comprehension of a program and on the ability to comprehend specific types of information. From this study of navigation and comprehension we make suggestions about how to aid novices in achieving high program comprehension.

The organization of this paper is as follows. Section 2 reviews the literature on navigation and comprehension of programs by novice programmers, then states our research questions. Section 3 describes the methodology. Our results are presented in Section 4. Section 5 discusses the results in relation to the research questions and prior research. Finally, Section 6 discusses the implications of the study and offers directions for the continuation of this research.

0-7695-1131-7/01 $10.00 © 2001 IEEE

2 BACKGROUND TO THE RESEARCH

Program Navigation

Before manipulating the source code of a completed program, the programmer must collect information about the program itself and then organize the information in some meaningful way. Based on the collection process and on the organized information, the programmer is then ready to carry out modifications, whether for debugging or for the enhancement of the program. These same steps can be used for the purpose of providing an understanding of the operation of the program without necessarily having the express goal of changing the source code. The process of collecting information about a program is referred to as navigation. During navigation the programmer gathers information about pieces of the program. The programmer also gathers information that is useful in recognizing the current location and in determining how to find a particular section of a program or item of information, starting from the current location.

Research shows that in trying to understand a program both experienced programmers and novices tend to follow similar overall strategies [7, 8, 11]. They begin their task by reading through the program. Experienced programmers, however, tend to employ a reading strategy that is different from the novice strategy. The following paragraphs discuss three methods that programmers may use to navigate programs. These navigation methods may be thought of as strategies for reading a program with the goal of information gathering. The methods of navigation are sequential, control flow, and data flow.

The *sequential* method of program navigation tends to be used by novices [8, 11]. In this method programmers read a program sequentially, line-by-line in its physical order, as if they were reading a book, and construct their mental model of the program sequentially [8, 11]. As a result, they tend to gain a fragmentary knowledge of the program that contains low-level knowledge but fails to connect the pieces in terms of their hierarchical structure, dynamic behavior, interactions, and purpose [8]. For example, the structure of a program written in Pascal places the main section of the program last, and all of the supporting procedures in the program come before the main section. Using a sequential reading strategy, novices will view the program in a bottom-up fashion, i.e., details first, general structure last. This is in contrast to the top-down approach of experienced programmers. In other programming languages, such as C, the order of the appearance of the main section of the program and the supporting procedures or functions is arbitrary. With the advent of function prototypes, whatever few previous constraints existed on the order of the program components have been effectively eliminated. In Pascal at least a loose precedence exists that requires a procedure to be declared before it is used. In C the programmer cannot depend on this relationship as a source of information when reading a program from beginning to end.

Although a sequential strategy is argued to be typical of novices, a variation of it is observed in experienced programmers under some conditions. For example, when a program is too complex for the programmer to gain a systematic understanding or when the programmer wishes to avoid the time it would take, a strategy of focusing on one sequential section at a time may be used. This has been referred to as an isolation strategy [11], as-needed strategy [10], or opportunistic relevance strategy [9].

The *control flow* method of program navigation is used by experienced programmers, who tend to follow the control flow in a top-down fashion just as the computer does when executing a program [8, 11]. Programmers begin with the first statement of the main section of the program and move to the next statement. Experienced programmers step into the procedures as they are encountered and continue with the steps found there. When a procedure is completed, the experienced programmer returns to the point in the program where the procedure was called and continues from there. Using this strategy, the programmer gains a hierarchical view of the program, in which sections of the program are viewed as units or "chunks" of information. These chunks encapsulate related statements and provide the advantage of making the information easier to manage than sequential, undifferentiated information that eventually becomes overwhelming. Soloway and his colleagues [16, 17, 18] have shown the importance of stereotypical chunks of functionally related statements in program comprehension. Wills [24] has used the idea of stereotypical chunks, which she refers to as clichés, in designing an automated tool for reverse engineering.

The *data flow* method of navigation may be used when the programmer is interested in one or more variables and how they are transformed throughout the program. Knowledge of data flow is required when the programmer wishes to modify a program to do a new calculation or to add additional variables in a calculation. The programmer now becomes concerned with understanding how the variables were calculated before being used and how the results of the calculation will be used later in the program. This requires tracing variables through the program. Navigating by following the data flow of one or more variables is certainly feasible but to our knowledge there are no existing studies of programmers using data flow navigation. On the other hand, empirical studies of programmers reading programs using their spontaneous navigation method have shown that data flow information is difficult to extract from procedural programs [13] and is extracted very little by novices [4]. Research on data flow notations [6] has shown that a notation that highlights data flow leads to better comprehension of data flow information. This suggests that systematic use of a data flow navigation method in a procedural program might aid in gathering data flow information from a program.

Mental Representations and Program Comprehension

The programmer may extract different kinds of information while reading a program. The programmer assimilates and organizes the information to form an abstraction of the program that we refer to as the mental representation, or mental model, of the program. The mental representation may be used as a source of knowledge for carrying out tasks with the program, for example, answering questions or modifying the program. Pennington's [12, 13] well known work on mental representation in programming identifies four basic categories of program information making up the programmer's mental representation: elementary operations in the code, control flow, data flow, and program goals. In her model, elementary operations and control flow information represent procedural aspects of a program. Program goals and data flow information together make up the representation of the situation described in the program. Pennington's model is minimalist in terms of the kinds of information she considers to make up the mental representation. Other researchers [2, 3, 20, 21, 22, 23] have supplemented Pennington's model to include additional kinds of information that may be found in programs, such as knowledge about program modules which occur in a larger program.

In this research we are interested in how the method of navigation affects the ability to extract different categories of information from a program. We base the categories of information on Pennington's model with additions consistent with the work of others [2, 3, 21, 23]. The result is five categories of information, as described below, which cover a broad spectrum of program relations important in program understanding.

- *Module sequential flow.* Module sequential flow concerns the <u>physical</u> ordering of the program's functions as the program is read from top to bottom. A comprehension question about module sequential flow might take the form: Is FUNC1 defined before FUNC2?
- *Control flow.* Control flow is the order of execution of the program, so it represents the logical order of statements in a program rather than the physical order. A comprehension question about control flow might take the form: When VAR1 and VAR2 are found to be equal, is an error message printed?
- *Data flow.* Data flow concerns how variables are manipulated in a program, that is, the transformation of variables from their initial state to their final state during execution. A comprehension question about data flow might take the form: Does the value of VAR1 affect the value of VAR2?
- *Global goals.* Global goals are the goals of the program as a whole. A comprehension question about global goals might take the form: Does the program convert the input currency into two different currencies for output?
- *Operations.* Elementary operations are the basic text units usually consisting of one or a few lines of code. A comprehension question about elementary operations might take the form: Is VAR1 set to 0?

Pennington [12, 13] and Burkhardt et al. [2] classify their individual knowledge categories into a program model, dealing with text-based knowledge, and a domain model, dealing with knowledge of the situation modelled in the program. Von Mayrhauser and Vans [21] posit a program model, a domain model, and a linking model spanning the other two. We do not use these distinctions in the current work because these groupings are likely to obscure the effects of navigation method on comprehension. For example, elementary operations and control flow may both be considered text-based information, but they may not both be facilitated by the same navigation method.

Research Questions

In this experiment we address two research questions concerning the navigation and comprehension of computer programs by novice programmers. These questions are discussed below.

The first research question concerns the global effect of navigation method on comprehension by novice programmers. Considering all categories of program information, we ask whether any navigation method gives an overall comprehension advantage. We hypothesize that control flow navigation will be globally most effective for program comprehension. The reasons for this prediction are two-fold. First, past research [8] has suggested that reading a program in control flow order leads to a top-down, hierarchical view of the program in which information is well-connected. By contrast, a sequential reading of a program leads to a more fragmented, bottom-up view that does not facilitate integrating the separate pieces of information into a coherent structure. Second, Pennington [13] argues that text-based relations, in particular control flow, are extracted from a program before domain-based relations, such as program goals and data flow. This argument is based on a theory that a domain-based representation is built up from text-based knowledge along with the external knowledge of the domain that the reader brings to the program. If it is the case that text-based control flow knowledge is necessary for understanding of domain-based relations, then a control flow navigation method would be preferred since it would facilitate control flow knowledge directly and also provide the basis for the programmer to understand the more abstract domain-based knowledge.

The second research question concerns the effect of navigation method on understanding of individual categories of information. Although we predict a global advantage for control flow navigation, it may be the case that certain navigation methods aid the comprehension of particular information in a program. We predict that a data flow navigation method, which allows readers to trace the relationships and transformations of variables throughout the program, will facilitate comprehension of data flow in the program. Sequential navigation emphasizes the top-to-bottom, physical order of the program, so it is expected to aid the comprehension of physical sequence information, such as the sequence of modules. While control flow navigation is expected to be globally superior, as argued above, we also expect that it will particularly facilitate comprehension of control flow relations in the program because the reading method follows, and thus highlights, the logical flow of the program.

3 METHOD

Experimental Design

A two-way mixed between-within design was used. The between-subjects factor was navigation method, with three alternative methods: sequential, control flow, and data flow. The within subjects factor was comprehension category with five information categories: module sequential, control flow, data flow, global goals, and operations. The dependent variable was the correctness of response to comprehension questions about the program.

Participants

The participants were 101 student programmers recruited from the first programming course in two different universities. The courses in the two universities were equivalent, emphasizing the basics of procedural programming, with an introduction to object-oriented programming later in the course. Participation in the study was voluntary. Seventy-one of the students were male and thirty were female. The average age of the participants was twenty-one years old. The majority were in their second or third year of university studies. Half of the participants had taken one course in high school that involved computer programming. On average, a participant had written about ten computer programs. The largest number of lines in any one program they had written, on average, was about one hundred lines. All participants were familiar with the C++ programming language, and especially the procedural C subset, because it was the computer programming language used in the courses in which the students were enrolled. Participants had completed at least three-quarters of the course before participating in the study but had not finished the course. All participants were familiar with a PC environment and with at least one web browser.

Materials

A training program and two experimental programs, were written in the C++ programming language utilizing minimal object-oriented features and emphasizing procedural aspects. The programs demonstrated good programming style, a structured approach, appropriate indentation, descriptive variable and function names, and use of standard constructs, i.e., the features that one would commonly find in a first programming course. The experimental programs did not contain comments in order to avoid giving the participant hints about the purpose and operation of the programs.

The training program was approximately 80 lines in length. The program accepted a Fahrenheit temperature, converted it to Celsius, and printed the result. The currency conversion program was about 360 lines in length and converted an amount of money from one currency to another (e.g., United States dollars to French francs). The discounts program was 310 lines long and calculated discounts, shipping, and total cost of items based on the number of items that were purchased.

Three sets of yes/no questions were written to test the participants' comprehension of the programs. One set consisted of 6 sample questions for use with the training program. They provided the participant with an idea of the kind of questions that would be asked after navigating a program. The remaining two sets of test questions were designed to cover the comprehension categories described previously. Six questions were written for each of the five comprehension categories. Of the six questions in each category, three questions would be answered correctly with a "yes" answer, and three questions would be answered correctly with a "no" answer. The questions were independent of each other. That is, the "no" questions were not converse restatements of the "yes" questions.

To evaluate the level of difficulty of the programs and questions, two professors who teach university-level programming courses using C++ independently analyzed the materials. Several enhancements were suggested and incorporated into the experimental programs. For the questions, they considered such aspects as similarity between sets of questions, level of difficulty (not too simple, but not too esoteric), relative difficulty between categories, comparison between "yes" and "no" questions, effort required to answer questions in a category, and whether the questions covered the whole program. Resulting suggestions for improving the homogeneity of the questions were implemented. The materials were later pilot tested by students from the target population.

The display and navigation of the program, as well as the question answering activity, were controlled by a computer program. A window similar to a simple text editor was used to display a program and allow the participant to navigate but restrict the participant to a specific navigation method. The window where the program was navigated displayed up to 24 lines with 80 characters per line. This window was the same size for all navigation conditions.

In sequential navigation, the participant was only allowed to scroll forward by either clicking on the down arrow in the scrollbar (move by single lines) or by clicking on the scrollbar below the elevator (move by half-screens or 12 lines). If the participant clicked on the up arrow in the scrollbar or in the scrollbar itself above the elevator, no action occurred. After the participant reached the end of the program, a Top button became active. By clicking on the Top button, the participant could return to the top of the program and begin sequential navigation through the program again. A set of blank lines was inserted between each function to separate one function from the next so that no two functions were displayed on the screen at the same time. As soon as the last line of a function scrolled off the top of the screen, the first line of the next function was visible at the bottom of the screen. This equalized the view with the other navigation methods where it was not possible to view more than one function at a time.

Participants using control flow navigation could scroll forward and backward, but the scope of the view was limited to the current function. The view of the text was similar to a web page with hypertext links. Function names were highlighted in red and also underlined. As the cursor passed over a function name, it changed from an arrow shape to a pointing finger. If the participant clicked on a function name, that function was displayed, and a Back button at the bottom of the window was activated. The participant could scroll through the current function, click on other function names within the current function to visit them, or click the Back button to return to the previous function. By being able to descend into functions in the same manner that the functions are logically related and executed, the participant was presented with a hierarchical view of the program.

The focus of the data flow navigation method is on the variables and how they affect each other. As with the control flow navigation display, the participant could scroll forward or backward through the program statements, but the scope of the display was limited to the current function. When the cursor moved over a variable name anywhere in the window, the cursor changed from an arrow to a pointing finger. The participant could then click on a variable name to highlight its occurrence throughout the current function. This calls attention to the places where the variable is used, either to be assigned a value or to provide a value. The participant could also right-click on a variable name in the display to produce a shortcut menu listing other variables in the program that are directly affected by this variable. If the participant selected one of the variables in the menu, the display was moved to the appropriate function and the variable was highlighted where it occurred in the function. The Back button was activated so that the participant could return to the prior function when finished with the current function. At the bottom of the display window was a drop-down list of all of the variables in the program. Instead of following a variable's path through the program, the participant could select one of the variables in the program from the drop-down list and click the Go button. The display was then positioned at the function where the variable occurred, and the variable was highlighted where it appeared in the function. This allowed participants to move directly to a variable they were interested in. This direct access to data elements via links was provided because traversing the often many links to get back to a particular variable one had seen previously was much more tedious than traversing the sequential or control flow links to get to a specific module.

Procedure

Participants were assigned randomly to navigation methods. Some of the original volunteers were eliminated for various reasons, such as equipment breakdowns or failure of the participant to complete the question set. Of the seventy-six participants who successfully completed the experiment, twenty-four were in the sequential navigation method, thirty in the control flow method, and twenty-two in the data flow method.

Testing was conducted in closed laboratories at the two universities during a regular class period. One laboratory contained twenty PCs running Windows 95 and the other contained thirty PCs running Windows NT, as well as the software needed to run the experiment. The students were familiar with the laboratory environment, the hardware, and the operating system. Participants were placed directly in the experimental software and did not interact with the operating system or other software. Participants were randomly assigned to navigation methods. The order of presentation for the two experimental programs was counterbalanced.

The core of the study was grouped into three identical phases, which were further divided into two activities. The first activity was the navigation of a program. The second activity was the completion of the comprehension questions for the program just navigated.

The first phase was the training phase. The training program was displayed along with program comments describing how to use the interface to navigate the program. The participant had approximately four minutes to explore the interface and navigate the training program. At the end of that time period, the program display window automatically closed, and the question activity began. Questions were displayed on the screen one at a time. The participant used the keyboard to answer the questions, pressing F for a "yes" response and J for a "no" response.

The participant had fifteen seconds to respond to a question. If a response was not received by that time, the computer recorded a "no answer" and continued to the next question.

The remaining two phases followed the same pattern. The experimental program was displayed, and participants had twelve minutes to navigate and read the program. At the end of the time period, the display window automatically closed, and the question activity followed. The comprehension questions were displayed in random order. As in the training phase, participants had fifteen seconds to respond to a question as in the training phase. At the end of the question activity, participants waited until everyone had completed the questions. The remaining experimental program was presented in the third phase, following the same display and questioning procedure.

4 RESULTS

The analysis was based on the correctness of responses to comprehension questions. Before the experiment, it was decided to analyze only the questions for which the correct answer was "yes." In forced choice questioning, "no" questions often are answered less correctly and more slowly than "yes" questions [15]. The cognitive processes involved in answering "yes" and "no" questions are different. "Yes" questions require a person to retrieve existing items of information from the mental representation. It is essentially a verification process, and the participant can answer "yes" as soon as the information is found. To answer a "no" question requires not verification but elimination. A person cannot answer "no" until an exhaustive search of the mental representation has been completed. In order to study a homogenous cognitive process, we analyzed only the "yes" questions. This decision is based on precedents in psychological research [15].

We combined the currency and discount programs for the analysis. The preliminary analysis showed that the discounts program questions were answered significantly more correctly than the currency program questions (F(1, 200) = 9.478, p < .003). However, there was no interaction between program and navigation method. Since navigation method was our main focus it was decided, for simplicity, to combine the results for the two programs.

We carried out a two-way mixed model Analysis of Variance. Navigation method was the between subjects factor, and comprehension category was the within subjects factor. The dependent variable was percentage of "yes" questions answered correctly. Our statistical procedures took into account the uneven number of participants per group. The main effect of navigation method was significant (F(2, 73) = 4.496, p < .014). The main effect of comprehension category was significant (F(4, 292) = 26.358, p < .001). The interaction between navigation method and comprehension category was also significant (F(8, 292) = 2.015, p < .045). Figure 1 shows the means of navigation method by comprehension category.

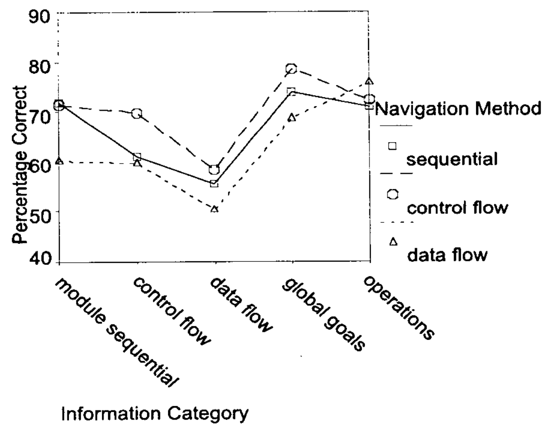

Figure 1 Means of the navigation methods in the five information categories.

To follow-up on the significant results, we carried out post hoc Tukey HSD tests. For navigation method, it was found that the mean of the control flow navigation method was significantly higher than the mean of the data flow method (p < .01). The mean of the sequential method fell between the control flow and data flow means. It did not differ statistically from the control flow or the data flow mean.

The significant interactions of navigation method with comprehension category are shown graphically in Figures 2, 3, and 4. The follow-up testing showed that in the module sequential category the scores for the sequential and control flow navigation methods were significantly higher than the scores for the data flow method (p < .05). In the control flow category the difference between the control flow and the data flow method approached but did not reach significance (p < .06), while there were no significant differences between the sequential method and the other two methods. Finally, in the global goals category, the scores for the control flow method were significantly higher than the scores of the data flow method (p < .05), but there was no significant difference between the sequential method and either of the other methods.

5 DISCUSSION

The data flow navigation method produced the lowest overall correctness means compared to the other navigation methods. This indicates that a view of the program highlighting variables and their transformations does not lead to good overall understanding of programs by novice programmers. There is a possible explanation o this result

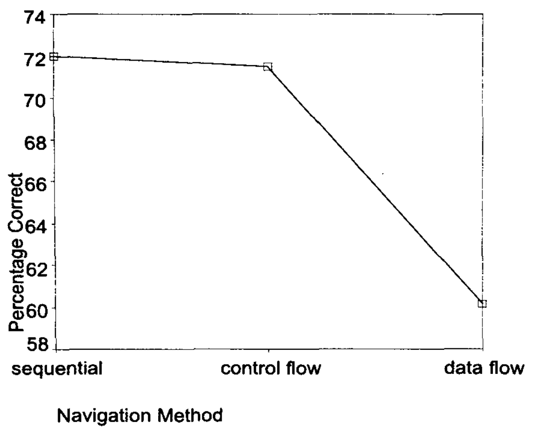

Figure 2 Means of the navigation methods for the module sequential category.

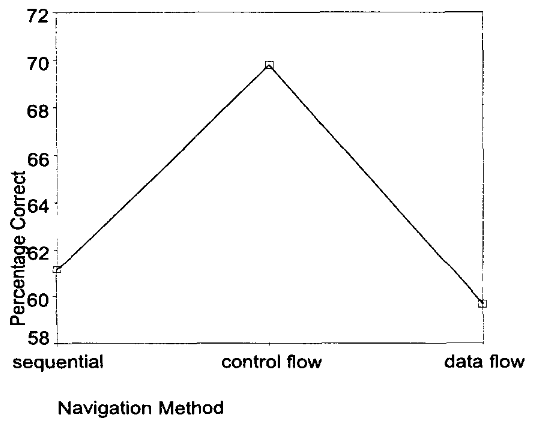

Figure 3 Means of the navigation methods for the control follow category.

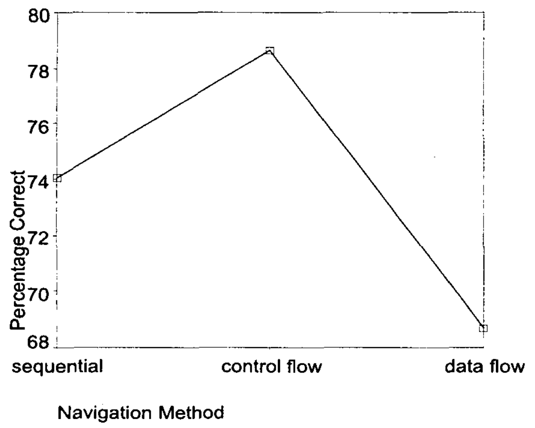

Figure 4 Means of the navigation methods for the global goals category.

in terms of theories of program comprehension. Pennington's [12, 13] bottom-up theory of program comprehension argues that programmers focus first on the textbase of the program, i.e. the basic structural entities. A programmer begins by gaining an understanding of elementary operations and control flow between operations. A mental representation of these basic text structures is built automatically while reading. Later, with continued exposure to the program, the programmer *may* build a mental representation of data flow and program goals, using knowledge of the basic text structures along with his or her domain knowledge. However, data flow and program goals are seen as higher-order knowledge that depends on having already formed a textbase. In this theory of comprehension, a navigation method or a view of a program highlighting data flow would not lead to development of a good mental representation because it does not aid the programmer in understanding the basic text structure. In fact, it may be argued that it would obscure that structure by emphasizing the transformation of variables. To develop a textbase, a programmer would need to ignore the data flow elements highlighted in the representation.

Von Mayrhauser and Vans' model of program comprehension [20, 21, 22] posits that a programmer may build the textbase and domain model simultaneously. This implies that a program view highlighting data flow could be valuable at the same time as views highlighting the program's textbase. The von Mayrhauser and Vans model is an expansion and generalization of Pennington's model taking into account the effects of expertise and program size. To develop a domain model early in comprehension of an unknown program requires substantial programming knowledge in long-term memory, particularly knowledge of clichés [24] concerning algorithms and data structures. Novices lack such knowledge and work primarily on developing a textbase, for which a data flow view is not very useful. However, experienced programmers might behave and use tools quite differently.

After the study, data flow participants said that data flow navigation was "hard", and that they "could not see all of the program" in a way that they wanted. That is, they were frustrated in being constrained to follow the data flow and did not feel that they were viewing the key information that they needed for comprehension of the program in the order that would be most helpful. They expressed a desire to be able to scroll through the program sequentially or to follow the control flow. Thus, it appears that the data flow order of reading was not useful in overall comprehension. Data flow participants could visit all the code, just as the sequential and control flow participants could. However, the strict adherence to a data flow view apparently obscured control flow and sequential structures that were essential. Despite their difficulty in making effective use of a data flow navigation method, participants were still able to construct some portion of a representation or understanding of the program, as shown by their correct response to 63.5 percent of the questions.

Interestingly, the data flow method of navigation did not even aid the comprehension of data flow information in the program, as shown by the lack of difference of the three navigation methods on the data flow questions. The data flow navigation participants did not perform any better on data flow questions than they did on other questions. They made more errors than any other group. In addition, data flow navigation resulted in poorer performance on questions about program goals, even though the literature on program comprehension argues for a strong link between data flow and program goals [12, 13]. These are signs that the novices were not able to make use of a data flow view of the program as a primary view. Data transformations are often spread across modules of a program. The programmer is required to find the different sections and make the mental connections between them to understand the significance of the data transformations [14, 19]. This normally requires a considerable amount of effort to accomplish. In principle, data flow navigation could greatly facilitate this effort, but if the programmer does not already have a sufficient textbase, then a view that moves from one section of the program to another following the manipulation of a variable may cause the programmer to lose the context of the manipulations.

The correctness scores means of the sequential and control flow methods of navigation were significantly different from the mean of the data flow navigation. However, there was no significant difference between sequential and control flow methods.

According to Jeffries [8] and Nanja and Cook [11], novice programmers' normal method of reading a program is sequential. Jeffries argures that novices spontaneously adopt a sequential method of reading because of their familiarity with reading sequential texts, such as books and magazines. The fact that participants using control flow navigation in our study did not significantly outperform participants using the sequential method of navigation may indicate that the novices are not yet highly skilled in using control flow information to construct an appropriate mental representation [8]. However, it should be pointed out that, even without explicit training in control flow navigation prior to the experiment, it was found to be just as effective as the novices' more familiar sequential method, indicating that participants are able to gather information using it. The control flow view did not strongly aid in comprehension of control flow relationships, as shown by the weak interaction of navigation method and comprehension category. However, it was the superior method for answering questions about program goals, or function. This is important because program goals contain information about what the program does, not just about how it does it. Interestingly the control flow method was as good as the sequential method for answering module sequential questions.

After the experiment, participants using the control flow method of navigation said that it was easy to navigate the program. They liked that method of navigation, and they were among the first to finish reading the program. Even though they were not trained in reading a program in this manner, they found it natural, sensible, and non-constraining. In their own way, the constraints on order of viewing of the control flow method were as strict as those of the data flow view. However, the control flow participants were able to gather information effectively within those constraints and achieve the highest overall scores.

Jeffries found that sequential reading led to a poor mental representation of a program in that novices failed to get a well-connected hierarchical view of the program. However, we found that the mental representation of these participants was equivalent to that of control flow participants. In our experiment, sequential navigation participants were observed to meticulously scan each line of the program before moving on to the next. They would often sit for a period of time reading through the lines of the function displayed in the window before moving on. This is typical of the novice programmer as observed by Jeffries [8]. A second approach observed was for participants to try to overcome the limitations of the navigation method. If the participant wanted to see a function previously encountered, the participant would move sequentially to the end of the program, then click the Top button and scroll forward to the desired function. This suggests that at least some novices wanted to cross reference what they were reading with what they had already read. However, their use of scrolling seemed more directed toward cross checking code already seen than on creating other non-sequential views of the program. Some sequential participants complained about not being able to move backwards in the program, but not about difficulty in following the control flow. Sequential navigation did not result in better comprehension of sequential information, such as the physical module sequence. While the data flow method had poorer results in the module sequential question category, the control flow and sequential navigation methods were equivalent.

6 CONCLUSION

Our basic result is that novices have great difficulty reading a program using a data flow view. Following data flow through a program seems chaotic and confusing to them. We relate this to the lack of highlighting of basic text structure units that, according to Pennington [12, 13], are the fundamental building blocks of program understanding. Our observations suggest that Jeffries [8] is correct in identifying the primary method of novice program reading as sequential. This conclusion is based on the high performance of sequential navigation participants in spite of its limitations. However, one of the most interesting results to us was the equally high performance of the control flow participants. Although they had not been trained in their computer programming course to read a program by following the control flow, they used this method successfully in the experiment. They also expressed a high level of comfort with it, and their comments lacked complaints about the restrictions of the method. This suggests the value of control flow navigation to novices. It does not appear that, as Jeffries [8] argues, novices are confused by the physical versus the logical order of the program. In fact, with appropriate training in the use of control flow navigation, novices might be able to make even better use of it to achieve high program comprehension.

We believe, based on Pennington's theory of program comprehension and our results, that a navigation method that highlights aspects of the textbase will aid novice programmers' comprehension. Sequential and control flow navigation highlight different aspects of the textbase. The facility to traverse a program with ease in both ways is fundamental. Once the textbase is understood sufficiently, then a data flow view of a program may be more valuable.

Our results lead us to pose additional questions about control flow navigation. The fact that novices did well with the method, in spite of lack of experience or training, suggests that the method may have strong potential to aid novice comprehension. We are interested in teaching novices how to read programs. We hypothesize that systematically training novices to follow the control flow of a program while reading may increase their comprehension, compared to simply allowing them to adopt an ad hoc method that may in effect be mostly sequential. In future work we intend to explore this hypothesis experimentally.

It should be noted that this study dealt only with small programs and novice programmers. Larger programs might have revealed a difference in comprehension between sequential and control flow navigation, since it may be harder to build sequentially a mental representation of a larger program. Experienced programmers have certainly developed ad hoc methods. From the work of van Mayrhauser and Vans [20, 21, 22] and our observations of experienced programmers in comprehension-demanding tasks [1], we believe that the methods of experienced programmers are a combination of the methods studied here, but also that programmers are quite variable in their approach to reading a program. We have also observed that they are highly variable in the ways in which they use tools to create different views of a program. The variability seems to come from many sources, including the size of the program, the domain of the program, the programmer's level of experience with programming in general and in the domain, the programmer's experience with the programming environment, the task to be carried out, and individual habits and preferences. Systematic study of these variables is needed to determine where the greatest improvements in program reading can be achieved.

REFERENCES

1. Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (1998). The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. *IWPC'98: Proceedings of the Sixth International Workshop on Program Comprehension*, (pp. 82-89). New York: IEEE.
2. Burkhardt, J.-M., Détienne, F. & Wiedenbeck,, S. (1997). Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension. *Human-Computer Interaction INTERACT'97*, 339-346.
3. Corritore, C. L. & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50, 61-83.
4. Corritore, C. L. & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3, 199-222.
5. Gilmore, D. J. & Green, T. R. G. (1984). Comprehension and recall of miniature programs. International Journal of Man-Machine Studies, 21, 31-48.
6. Green, T. R. G., Petre, M., & Bellamy, R. K. E. Comprehensibility of visual and textual programs: a test of superlativism against the 'match-mismatch' conjecture, in: Koenemann-Belliveau, J., Moher, T.G., and Robertson, S. P. (Eds.), *Empirical Studies of Programmers: Fourth Workshop* (Ablex, Norwood, NJ, 1991) 121-146.
7. 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.
8. Jeffries, R. (1982). A Comparison of the Debugging Behavior of Expert and Novice Programmers. A paper presented at the AERA Annual Meeting.
9. Koenemann, J. & Robertson, S. P. (1991). Expert Problem Solving Strategies for Program Comprehension. *Communications of the ACM*, 34, 4, 125-130.

10. 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: First Workshop.* Norwood, NJ: Ablex.

11. 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, 172-183.

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

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

14. Pennington, N. & Grabowski, B. (1990). The Tasks of Programming. In J. Hoc, T. Green, R. Samurcay & D. Gilmore, Eds. *Psychology of Programming.* London: Academic Press.

15. Rosch, E. (1975). Cognitive representation of semantic categories. *Journal of Experimental Psychology: General*, 4(3), 192-233.

16. 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.

17. Soloway, E., Bonar, J. & Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26, 11, 853-860.

18. Soloway, E. & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10, 5, 595-609.

19. Soloway, E., Pinto, J., Letovsky, S., Littman, D. & Lampert, R. (1988). Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, 31, 11, 1259-1267.

20. von Mayrhauser, A. and Vans, A. M. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6), 424-437.

21. von Mayrhauser, A. and Vans, A. M. (1995). Program understanding: Models and experiments. In M. C. Yovits and M. V. Zelkowitz (Eds.), *Advances in Computers*, vol. 40, pp. 1-38. Academic Press, San Diego.

22. von Mayrhauser, A. and Vans, A. M. (1997). Program understanding needs during corrective maintenance of large scale software. *COMPSAC 97: Proceedings of the 21st Annual International Computer Software and Applications Conference*, pp. 630-637, IEEE Computer Society Press, Santa Monica, CA.

23. Wiedenbeck, S., Fix, V. & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39, 793-812.

24. Wills, L.M. (1993). Flexible control for program recognition. *Proceedings of the First Working Conference on Reverse Engineering.* Baltimore, MD, pp. 134-143, IEEE Computer Society Press, Santa Monica, CA.