Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study

Cynthia L. Corritore
College of Business Administration
Creighton University
Omaha, NE 68178 USA
cindy@creighton.edu

Susan Wiedenbeck
College of Information Science & Technology
University of Nebraska
Omaha, NE 68182 USA
Susan_Wiedenbeck@unomaha.edu

Abstract

This study examines the direction and scope of comprehension-related activities of professional programmers carrying out several comprehension and maintenance activities over time. Procedural and object-oriented (OO) programmers studied a program and subsequently performed modifications during two sessions. Results showed that the OO programmers tended to use a strongly top-down approach to program understanding during an early phase of study of the program but increasingly used a bottom-up approach during the maintenance tasks. The procedural programmers used a more bottom-up orientation throughout all activities. The scope of the activities was greater for the procedural than for the OO programmers. However, regardless of paradigm, the programmers over time built a broad, rather than a localized, view of the program.

1. Introduction

The comprehension of program code is one of the key cognitive activities in programming, and program maintenance depends, in part, on comprehension [6, 20, 26, 27, 28]. To make successful modifications to a program without introducing errors, the programmer must have an adequate understanding of what a program does and how it does it. However, the question is what "adequate" means. To answer that question requires experimental work. To date, several experimental studies of programmers carrying out maintenance tasks [15, 17, 18, 29, 30] have opened up this field to study. Now there is a need for more knowledge about what information sources programmers use to understand programs during maintenance activities, how the use of the information sources evolves over time, and what effect, if any, the programming paradigm has on program understanding during maintenance.

In this study we track the knowledge sources used by expert programmers during maintenance of a program in order to determine: 1) the scope of comprehension, and 2) the direction of comprehension. In addition, we analyze how comprehension-related activities evolve in successive maintenance episodes on the same program and how the information gathering and comprehension of procedural and OO programmers differ. This study differs from earlier studies in comparing the activities of procedural and OO programmers. It also uses larger programs than earlier studies and makes observations of activities in several comprehension and maintenance episodes over a longer period of time.

2. Prior research on comprehension

2.1 Scope and direction of comprehension activities

The **scope of comprehension activities** refers to the breadth of familiarity with the program gained by the programmer during comprehension activities. Littman et al. [17] found two strategies used by programmers concerning scope of comprehension, systematic and as-needed. Using the systematic strategy, the programmer attempts to gain a broad understanding of the program before carrying out modifications. The goal is to understand the design of the original programmer so that modifications fit with the existing code. On the other hand, using the as-needed strategy, the programmer attempts to minimize the amount of code that has to be understood. The programmer does not

0-7695-0656-9/00 $10.00 © 2000 IEEE

attempt to understand the overall design of the program but concentrates instead on the functioning of selected local parts of the code that are critically involved in the modification. Littman et al. found that programmers who used the systematic approach carried out modifications more successfully. The authors argue that programmers using the systematic strategy were more successful because their systematic study increased their ability to detect interactions between the code central to the modification and code elsewhere in the program.

While a systematic comprehension strategy may be most successful for making modifications in the laboratory, it is not clear how usable such a strategy is outside the laboratory. von Mayrhauser and Vans [27] present the view that systematic comprehension may be infeasible in a large program with thousands of lines of code. An experiment by Koenemann and Robertson [15] made a similar argument about scalability and began to address comprehension strategies in larger programs. Koenemann and Robertson found that comprehension was as-needed in a larger program and that programmers did not attempt to use a truly systematic strategy. Instead, the scope of comprehension was largely directed by the modification task being attempted. Browsing for identifiers was used to find possibly relevant procedure names, but not to gain an overview of the program. Programmers did not spend time comprehending parts of the program that they believed irrelevant to the modification task. On the other hand, von Mayrhauser and Vans [29] observed an instance of systematic study of an large program by a professional programmers in industry. However, it is not clear how large a part of the program was subjected to systematic study or whether this behavior was ever repeated.

The **direction of comprehension activities** concerns whether the strategic approach to program comprehension is top-down, bottom-up, or a mixture of the two. Shneiderman and Mayer [23] and also Pennington [19] have proposed largely bottom-up theories of program comprehension. For example, Pennington's [19] well-known comprehension model describes two program abstractions formed by the programmer during comprehension. The *program model* is a low-level abstraction consisting of detailed knowledge of operations at a level close to the program code and of control flow relations representing the order of execution. The program-level abstraction is formed early during program understanding using information in the program text. The *domain model* is a higher-level abstraction containing knowledge of data flow and function. This abstraction is formed after the program model. It is built from knowledge in the program model along with knowledge of relevant programming plans. Pennington's experimental studies [18, 19] and a study by Bergantz and Hassell [1] support the existence of these two program abstractions during comprehension and also the formation of the program model before the domain model.

Brooks [3] and Soloway and his colleagues [25] present top-down, hypothesis-driven theories of program understanding. For example, the theory of Soloway and his colleagues [24, 25] treats comprehension as a plan-based activity. Plans are schematic knowledge about how to carry out stereotypical actions in a program. Plans exist at different levels. Strategic plans are global plans for the solution of a problem; tactical plans are language-independent specifications of algorithms for solution of local parts of a problem; implementation plans are plans for carrying out a tactical plan in a given programming language. Also, discourse rules are programming conventions that govern how plans are expressed and combined. Program understanding begins with the programmer hypothesizing a high-level program goal, then decomposing it into more specific subgoals that should be present in a program having a given high-level goal. Having identified expected goals and subgoals, the programmer must determine whether they exist in the program. The programmer uses knowledge of stored plans and discourse rules to try to satisfy the subgoals and ultimately the top-level goal. Modification of the subgoals and iteration of comprehension is required if they cannot be directly satisfied by supporting evidence in the program. Soloway and his colleagues (cited above) present empirical support for their plan-based model.

While the top-down and bottom-up models have been very influential, today mixed models of program comprehension are increasingly viewed as more realistic descriptions of large program comprehension [16, 28, 29, 30]. These models consider programmers to behave opportunistically in program understanding, switching from top-down to bottom-up comprehension strategies depending on the situation. Shaft and Vessey [22] and von Mayrhauser and Vans [29] propose that programmers use a top-down, goal-oriented, or hypothesis-driven, approach to understanding when they are working in a familiar domain where they know a large number of plans. On the other hand, when they encounter code that is new to them and in an unfamiliar domain, they use the bottom-up approach described by Pennington, first developing a program model consisting of a control flow abstraction from the program text and later forming a domain model consisting of data flow and functional abstractions. In a

large program consisting of tens of thousands of lines of code or more, switches from top-down to bottom-up comprehension and vice versa may occur frequently because the programmer's knowledge about the domain varies in different parts of the program. von Mayrhauser and Vans [31] observed that the use of bottom-up and top-down comprehension varies with the task, and that it also varies with the familiarity of the programmer with the domain and the program at hand. Programmers with knowledge of the domain take a more top-down approach than do programmers with less domain knowledge. Also, programmers with less language knowledge or less knowledge of the current program are more bottom-up in comprehension [30].

2.2. Procedural and OO comprehension

Most of the possible advantages identified for the object-oriented paradigm concern design and reuse. However, von Mayrhauser and Vans [29] suggest that there may also be advantages for program comprehension if OO-style programs support a more top-down, domain-oriented approach to comprehension. They call for the extension of research on comprehension in program maintenance to programs written in the object-oriented paradigm. Gilmore and Green's [11] work on the effect of program notations on information availability suggests that a program style, or notation, will lead to better comprehension of given information if it highlights that information in some way. The OO style emphasizes objects, their actions, and their relationships and thereby highlights domain-based information. It may also be argued that the OO style tends to have more inherent code-level structure and may encourage more design structuring. Consistent with Gilmore and Green's argument, experimental results of Burkhardt, Détienne, and Wiedenbeck [4] show that OO experts do tend to develop a domain-based abstraction in terms of these entities during program comprehension. Further results [5] suggest that OO experts use more top-down behavior than do OO novices, while the scope of comprehension of OO experts and novices is similar. Corritore and Wiedenbeck [8] compared OO and procedural experts directly in program comprehension. They found that there was no difference in comprehension of function and data flow by the two groups, but OO programmers initially developed stronger knowledge of program structure including the relationships of program objects, and poorer knowledge of specific operations and control flow. This suggests a more top-down approach among OO programmers. However, after modifying the program, OO programmers improved their program-level knowledge to equal that of the procedural programmers.

Few experimental studies comparing maintenance in the OO and procedural paradigms exist. Henry and Humphrey [13] found that modifications to OO programs were more local, i.e., involved editing of fewer modules, than modifications to the corresponding procedural programs. This might indicate that it is possible to successfully make changes in an OO program using a local, as-needed comprehension strategy. Daly, Brooks, Miller, Roper, and Wood [10] studied the maintainability of OO programs as a function of the depth of the inheritance hierarchy. A deeper hierarchy led to problems in making modifications. These problems appear to be comprehension-related, e.g. difficulties tracing the inheritance hierarchy.

2.3. Research questions

Our research questions focus on comprehension strategies and how they change over time during program understanding and maintenance.

Our first question is whether OO experts show a more top-down direction of comprehension activities than procedural experts. The OO style with its greater structuring and emphasis on objects, their hierarchical relationships, and their functionality may facilitate a more top-down strategy.

Our second question is whether OO experts have a narrower scope of comprehension activities than procedural experts. In the OO paradigm, objects encapsulate both data and functionality. Because of encapsulation, changes to the functions of an object are internal to the object. This may mean that modifications can be done with less broad knowledge of the program as a whole.

Our third question is how comprehension strategies change over repeated modification of the same program. A narrow scope of comprehension activities in initial maintenance of a program may incrementally become a wide scope, as a programmer works with a program over time. Thus, we believe that it is necessary to begin considering program understanding and maintenance in a longitudinal perspective.

3. Research method

3.1 Participants

Thirty professional programmers participated in the study. Fifteen were procedural C programmers, and 15 were object-oriented C++ programmers. All but 2 had

post-baccalaureate degrees. Twenty-seven held their highest degree in computer science or engineering. On average, participants had been programming for 11.6 years with a range of 2.5 - 20 years. The average length of full-time employment as a system analyst or programmer was 7 years.

The C and C++ groups did not differ statistically with respect to demographics or programming experience. However, a known difference between the C and C++ groups was that the C group had relatively low exposure to OO programming, while most members of the C++ group did have previous experience programming in procedural languages. This difference between the groups results from the fact that currently most programmers with substantial experience in industry began their careers working in procedural languages.

3.2 Materials

C and C++ were chosen to represent the procedural and OO paradigms respectively because the language notations are the same except for the specifically object-oriented features, reducing the likelihood that extraneous differences between the languages would affect the results. Two functionally equivalent versions of a database program were written in procedural C and object-oriented C++. The program manipulated records of passengers, crew members, and flights of a small airline. This database problem was chosen because it did not require highly specialized domain knowledge.

Both programs and their supplementary materials were as equivalent as possible, while at the same time prototypical of their paradigm. The C++ program represented complex data entities through classes and made extensive use of the OO features of inheritance, composition, encapsulation, and polymorphism. The procedural program was written in a structured and modular manner. It implemented complex data entities through structure variables. Both programs used array and list data structures. In the C++ program, the functions were methods attached to classes, while in C stand-alone functions were used. The same underlying algorithms were used in corresponding functions of the C and C++ program (for searching, string processing, etc.). The two programs also used the same or similar variable names whenever possible. Both programs were similar in length, C++ 822 lines vs. C 783 lines. The C++ program was partitioned into files corresponding to objects. The C program was partitioned into files by the main conceptual entities (e.g., customer, flight). These corresponded to objects in the OO program. Each program also had standard C/C++ files, such as main and includes.

Comments were not included in the source code of either program; however, extensive external documentation was provided. Documentation to supplement the source code included program summaries, descriptions of the functionality of program modules, charts of inheritance hierarchies, charts of calling structures, and diagrams of data structures. The materials were designed to be functionally equivalent for the two paradigms, although they were not identical. The most notable difference was that there were no inheritance hierarchy charts for the procedural program. However, in place of inheritance hierarchy charts, a substitute documentation was provided for the procedural program that showed graphically the C structures and their components. This served to balance the amount of documentation and to provide information about the program entities. Two program modification tasks were also designed. The programs, associated external documentation, and modification tasks were inspected by C and C++ experts, who evaluated the similarity of the content of the materials and the typicality of the programs and tasks to their respective paradigms. Modifications were made based on their critiques.

3.3 Procedure

Each participant was run individually in two 2-hour sessions that were held seven to ten days apart. In the first session the participant studied the program for 30 minutes, followed by a short modification task. This modification was used to motivate the study of the program, but data were not collected on it. Participants completed the two experimental program modifications during the second session. The order of presentation of the modifications was counterbalanced.

The program and all supplementary materials were presented on-line in a graphical Unix environment created for the study. The environment supported standard editing features such as cut and paste and also compilation and running of the program. One restriction of the environment was that only one document (segment of program code or supplementary documentation) was visible at a time. This restriction allowed us to determine clearly what materials participants accessed. The participants were not familiar with the environment prior to the experiment. They were trained in its use at the beginning of the first session. Screen capture software recorded participants' use of the materials. Data were collected in the study phase and the two modifications. The data consisted of which files were accessed by participants but not the order in which files were accessed.

4. Results

For purposes of this study, we defined the scope of comprehension activities as the proportion of files accessed. A systematic strategy would be indicated by a broad study of *all* or almost all of the available program materials. An as-needed strategy would be indicated by limitation of the study of program materials to only a small part of those available. We defined the direction of comprehension activities as the level of abstraction of the program materials accessed. Our program materials were at three levels of abstraction, corresponding to three file types that we provided to the participants: documentation files (.doc), header files (.h), and implementation files (.cc). Documentation files were the most abstract, containing the external documentation of the program. Header files contained the declarations of data elements and functions and were at a middle level of abstraction. Implementation files contained the code of functions and were at the lowest level of abstraction. Accessing the more abstract documentation and header files was interpreted as reflecting the use of top-down processes and accessing the less abstract implementation files was seen as reflecting a bottom-up strategy. A mixed direction of comprehension, incorporating top-down and bottom-up elements, would be indicated by the use of files at different levels of abstraction.

Analysis of Variance was used to examine differences between procedural and OO participants in the proportion and type of files accessed during the different experimental tasks: program study, modification 1 and modification 2. Follow-up analysis was carried out using ANOVA and Tukey's HSD. Since our pool of participants was limited by the high expertise required, we had a small sample size, leading to reduced statistical power. As a result, we set .10 as our alpha level. We did follow-up analysis on the ANOVAs if: 1) the p-value was less than .10, *and* 2) the effect size (ε^2) was moderate or large by Cohen's (1977) guidelines, i.e., ε^2 of less than .06 is small, between .06 and .14 is moderate, greater than .14 is large. Effect size indicates the strength of a relationship and is often considered an indication of practical significance. As the tests below indicate, we found that a number of results were marginal in terms of statistical significance but had a large effect size.

The basic analysis was a three-way, mixed-model Analysis of Variance. The between-subjects factor was *programming paradigm* (procedural or object-oriented). The two within-subjects factors were *file type* (documentation, header, or implementation) and *activity* (program study, modification 1, or modification 2). The dependent variable was the mean proportion of files accessed. The range of the means of the dependent variable was 0 to 1.

The ANOVA indicated a significant main effect of paradigm ($F(1,28) = 4.03$, $p < .05$, $\varepsilon^2 = .13$). Procedural participants accessed significantly more files than OO participants overall. In addition, there was a significant main effect of file type ($F(2,56) = 7.89$, $p < .001$, ε^2 =of .22). The follow-up analysis indicated that, considering all participants and activities together, a higher proportion of implementation and header files was accessed than documentation files ($p < .05$), but there was no significant difference between accesses of header and implementation files. The main effect of activity was significant, as well, and had a large effect size ($F(2,56) = 34.09$, $p < .001$; $\varepsilon^2 = .55$). Follow-up analysis showed that all participants accessed more files during the study period than during the modifications ($p < .05$).

Table 1: Means and standard deviations (in parentheses) of proportion of files accessed by file type in the three activities.

	Implement. Files	Header Files	Docu-ment.	All
Study	.75 (.27)	.76 (.25)	.75 (.35)	.75 (.18)
First Mod.	.57 (.19)	.43 (.26)	.38 (.37)	.46 (.29)
Second Mod.	.55 (.21)	.45 (.33)	.15 (.23)	.39 (.31)
All	.62 (.14)	.55 (.15)	.43 (.24)	.53 (.08)

Table 2: Means and standard deviations (in parentheses) of proportion of files accessed by the procedural and object-oriented groups for the three file types.

	Procedural	OO	All
Documen-tation	.37 (.28)	.48 (.19)	.43 (.24)
Header Files	.60 (.13)	.49 (.16)	.55 (.15)
Implement. Files	.70 (.11)	.54 (.12)	.62 (.14)
All	.56 (.08)	.50 (.07)	.53 (.08)

There was a significant two-way interaction of file type and activity ($F(4,112) = 6.89$, $p < .001$, $\varepsilon^2 = .20$, see Table 1). There was also a significant interaction of paradigm and file type ($F(2,56) = 3.94$, $p < .03$, $\varepsilon^2 = .12$, see Table 2). However, the two-way paradigm by activity interaction was not significant.

Follow-up on the file type by activity interaction indicated that participants accessed documentation files more during the study period than during the modifications ($p < .05$). In addition, they accessed more header files during the study period than during the modifications ($p < .05$).

Follow-up testing for the paradigm by file type interaction did not find any significant differences, most likely because of low statistical power. However, it appears that there was a trend for procedural participants to access more implementation than documentation files over all three activities.

The three-way interaction among paradigm, activity, and file type was also significant ($F(4, 112) = 2.35$, $p < .058$, $\varepsilon^2 = .08$, see Table 3). In the follow-up analysis we used two-way ANOVA to examine each activity separately, using paradigm as the between subjects factor and file type as the within subjects factor. For pairwise comparisons Tukey's HSD was again used.

During the *study phase*, the main effects of paradigm and file type were not significant, but there was a significant interaction of paradigm and file type ($F(2, 56) = 7.28$, $p < .01$, $\varepsilon^2 = .21$). Pairwise comparisons showed that, during the study phase, the procedural participants accessed significantly more implementation files than did the OO participants ($p < .05$). In contrast, OO participants accessed significantly more documentation files than did procedural participants during the study phase ($p < .05$). Within the procedural group, implementation and header files were accessed significantly more than documentation files ($p < .05$). By contrast, within the OO group documentation files were accessed more than either header or implementation files during the study phase ($p < .05$).

Table 3: Means and standard deviations (in parentheses) of proportion of files accessed by the procedural and OO groups for the three file types in the three activities.

		Procedural	OO
Study Period	Doc. Files	.62 (.44)	.88 (.16)
	Header Files	.81 (.17)	.70 (.31)
	Impl. Files	.85 (.25)	.64 (.26)
Mod. 1	Doc. Files	.38 (.39)	.37 (.36)
	Header Files	.43 (.30)	.44 (.21)
	Impl. Files	.62 (.16)	.52 (.22)
Mod. 2	Doc. Files	.12 (.19)	.18 (.28)
	Header Files	.56 (.35)	.34 (.29)
	Impl. Files	.63 (.18)	.47 (.21)

In *modification 1*, a significant main effect of file type was discovered ($F(2, 56) = 3.56$, $p < .04$, $\varepsilon^2 = .11$). The main effect of paradigm was not significant, nor was the paradigm by file type interaction. Pairwise

comparisons indicated that, across both paradigms, participants accessed more implementation files than documentation files during modification 1 ($p < .05$).

During *modification 2*, there were significant main effects of file type ($F(2, 56) = 20.88$, $p < .001$, $\varepsilon^2 = .43$) and paradigm ($F(1, 28) = 3.39$, $p < .08$, $\varepsilon^2 = .11$). Procedural participants accessed more files overall during modification 2 then did the OO participants. Follow-up testing on the main effect of file type showed that all participants accessed more implementation files than header files and more header files than documentation files during modification 2 ($p < .05$). The ANOVA also indicated a significant two-way interaction of paradigm and file type ($F(2, 56) = 2.75$, $p < .07$, $\varepsilon^2 = .09$). Pairwise comparisons showed that the procedural group accessed more implementation and header files than documentation files during modification 2. Their use of documentation files was near zero. The OO group similarly accessed more implementation files than documentation files during modification 2 and showed a strong drop in use of abstract information.

While performance on the maintenance tasks, was not the focus of this study, it is interesting to note that the OO and procedural groups did not differ in this respect. The solutions were evaluated for correctness and completeness on a three point scale, and there was no significant difference between the groups.

5. Discussion

The scope of comprehension (measured as the proportion of files accessed) of participants in this study was quite broad. Our results show that a high proportion of files was accessed, and by the end of modification 2 participants were potentially aware of most of the code from having accessed such a large proportion of the files. This differs from the finding of Koenemann and Robertson [15] who report a very localized scope of comprehension. They explain the narrow scope of comprehension in their study by suggesting that programmers only look at code or documentation that they perceive as directly relevant to a programming task currently being carried out. However, our findings of high percentages of file accesses and access of almost all functions by the end of the second modification suggest a broader scope of comprehension. While programmers are certainly motivated in the short-term by the demands of the task at hand, a longer-term perspective, as in this experiment, shows that knowledge accumulates over different tasks on the same program, until a broad scope of comprehension is achieved. This growth of knowledge allows programmers to anticipate and deal with interactions which occur when a modification in one part of a program affects the code in another part.

Although it is generally true that the scope of comprehension was broad, the scope was influenced by the phase of the experiment and the programming paradigm. The experiment was divided into a study phase followed by two modification phases. Participants generally had a greater scope of access to files of all types while initially studying the program. During the two modifications, the scope became narrower and more focused, as reflected in lower file access rates. Although we did see a narrowing of access to files, comparing our procedural participants to those of Koenemann and Robertson [15], the breadth in our study remained much greater, i.e. access to 40 percent of the functions after the modifications in our study, as opposed to 20 percent of the functions in their study. This difference is probably related to greater complexity of the tasks in our study, our longer three phase design, and the nature of the program. Programmers in our study made modifications to a program that they had first studied and modified a week earlier. Because of the time lapse, there may have been some recurring study behaviors as they revisited parts of the program to make the modifications in the second session. We see this revisiting of code as realistic. In the workplace, maintenance programmers may have to return to code that they have modified previously, so they have an on-going need to re-familiarize themselves with a program. However, as in this experiment, the study of the program is somewhat different each time because of the motivation provided by the specific maintenance task.

In our experiment the procedural group had a greater scope of comprehension than the OO group. The difference was reliable, although not very large: 59 percent of the functions were visited by the procedural group vs. 40 percent of the functions by the OO group. The greater scope of the procedural group carried through the study phase and the two modifications. However, the explanation for the difference in scope may be different in the study vs. the modification tasks. In the study period, the scope and direction of comprehension strategies may have interacted. If the OO participants used a top-down approach in the study phase, they may have gained a broad overview of the program from the abstract documentation files without having to consult so many of the numerous but less abstract implementation files. In the later program modification tasks, the OO group may have had a narrower scope because the encapsulation of the OO paradigm facilitated the OO participants in focusing their efforts. That is, the encapsulation made the changes more local to a specific class and protected the

rest of the program from effects of the changes. This effect of encapsulation is predicted in the OO literature [9, 12, 14, 21].

We classify the direction of comprehension (measured as the level of abstraction of files accessed) in our experiment as "mixed." We observed substantial use of both top-down and bottom-up strategies. Nevertheless, the bottom-up orientation was globally more prevalent, as reflected by higher access rates to implementation and header files compared to documentation files. The documentation files were used heavily only during the study period. Furthermore, they were especially heavily used by OO participants. Thus, the direction of comprehension was influenced by both the phase of the experiment and the paradigm. During the study period, OO participants used a top-down strategy relying strongly on documentation: they accessed almost 90 percent of the documentation files but only 60 to 70 percent of the other files. Procedural participants used a more bottom-up strategy during the study period, accessing slightly over 60 percent of the documentation files but 85 to 90 percent of the header and implementation files. They continued to be bottom-up during the modifications maintaining higher access rates to the implementation and header files than to the documentation files. By contrast, the OO group changed their strategy more over time depending on the activity. Initially they used a top-down approach to comprehension, but in the later phases, as they made program modifications, their strategy became more bottom-up. This shift of orientation became more clear as the experiment progressed, and is most evident in the sharp decline in their use of documentation in the last modification. We suggest that the more abstract files provided the information necessary for general comprehension of the program, but a more bottom-up orientation was needed to gather all of the information needed to carry out the modifications. The demands of the modification tasks focused the information needs of the programmers on the code itself and, furthermore, on specific parts of the code. It is true that the same decline in the use of abstract information occurred in the procedural group; however, it was less extreme because they used external documentation fairly lightly throughout.

The use of a mixed strategy and the change over time suggest that the type of activity and the programmer's history of experience with the program play important roles in the direction of comprehension. In an initial phase of familiarization with the program, more top-down behavior is found. However, after the program is partially familiar and given the motivation of a modification task, abstract descriptions are less useful, so the strategy tends to be much more bottom-up. At the same time, the programmer is continuously gaining greater knowledge of the program in the course of carrying out modifications. This accumulation of knowledge about the program results in even less need for abstract documentation in later modification tasks. This agrees with results presented by von Mayrhauser and Vans [29]. It also suggests that theories of program comprehension should take into account at least the task motivation and the programmer's longitudinal experience with the program. They may also need to take into account the programming paradigm, since our results show that the OO group took a more top-down approach during initial study with the program than did the procedural group. In prior work, Burkhardt et al. [5] observed a top-down orientation in OO experts. Our results agree with Burkhardt et al.'s with respect to the early phase of study or familiarization with a program. However, we found a change to a bottom-up strategy with experience with a given program and motivated by a modification task.

6. Conclusion

Two characteristics of strategic importance in program understanding are the scope of comprehension and the direction of comprehension. This research provides evidence about the scope and direction of comprehension of professional procedural and OO programmers in understanding and modification of a program over repeated episodes.

Both the procedural and the OO group employed a wide scope of comprehension over the course of the study. The scope was greater during the initial study period, then narrowed during the modifications. However, it still remained relatively broad in both groups of programmers throughout the experiment. This was in contrast to the finding of a highly restricted scope of comprehension by Koenemann and Robertson [15]. From our experiment it appears that programmers do indeed restrict their breadth of comprehension of a program to focus on the parts most strongly relevant to the modification. However, at the same time they attempt to understand the program broadly enough to notice potential interactions with other program code. With respect to procedural vs. OO comparison, we found that the scope of comprehension was broader in the procedural than the OO group, particularly during the study period. This may indicate that information is more scattered in the procedural paradigm, forcing programmers to visit many implementation files to find what they need. However, since there are few studies comparing the OO and procedural paradigm, we feel that more experimental evidence is needed on this point

to make a firm conclusion.

The comprehension strategies of procedural and OO programmers also differed with respect to direction of comprehension. The differences between the two groups were strongest during initial program comprehension activities and were less evident during the program modifications. We conclude that both groups employed a comprehension strategy that was largely bottom-up when making successive modifications to the program. However, during the early phase of program comprehension, OO programmers tended to utilize a top-down orientation. Procedural programmers, on the other hand, were more strongly bottom-up even during this early phase.

A limitation of this study is the grain of the analysis. We used proportion of files accessed and level of abstraction of files accessed as surrogates for scope and direction of comprehension. From sets of comprehension questions administered in the middle and at the end of the experiment, we do know that the participants' understanding of the program increased over time and that they had a relatively good understanding of the program at the end of the experiment [8]. However, from counting file accesses we do not know exactly what the participants read or thought about while looking at a file. Furthermore, we know how use of files evolved over the three phases of the experiment, but within those phases we do not know the order in which files were accessed. More detailed studies, possibly using videotaping and a think aloud method, are needed to overcome these limitations.

This study makes a beginning at examining the effect of paradigm on comprehension. More focused research into the differences in comprehension between OO and procedural programmers is needed. In this study we faced the dilemma of experimental control vs. generalizability that so often arises in research on programming. We chose to carry out a laboratory experiment with high experimental control but at the expense of using a small program and relatively artificial setting. In our view, controlled laboratory experiments and workplace observational studies are complementary. We think that two kinds of studies are called for: laboratory studies using programs *as large as possible* for an experimental study, along with a *more extended longitudinal approach* of repeated exposure to the program, and also observational studies of programmers comprehending large programs in the workplace using a technique of *repeated data collection over time*. In workplace observations we would like to see, if possible, the use of techniques such as contextual inquiry [2], which allow the researcher to get a better understanding of the reasons for the programmers' actions, at the expense of slightly higher intervention in their work.

7. References

[1] 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.

[2] Beyer, H. and Holtzblatt, K. (1998). *Contextual Design: Defining Customer-Centered Systems.* San Francisco, CA: Morgan Kaufmann.

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

[4] Burkhardt, J-M, Détienne, F., and Wiedenbeck, S. (1997). Mental representations constructed by experts and novices in object-oriented program comprehension**,** *Proceedings of INTERACT'97: 6th IFIP International Conference on Human-Computer Interaction* (pp. 339-346). Amsterdam: North Holland.

[5] Burkhardt, J.-M., Détienne, F., and 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.

[6] Canfora, G., Mancini, L., and Tortorella, M. (1996). A workbench for program comprehension during software maintenance. *Proceedings of the Fourth Workshop on Program Comprehension*, (pp. 30-39). Los Alamitos, CA: IEEE Computer Society.

[7] Cohen, J. (1977). *Statistical Power Analysis for the Social Sciences*. New York: Academic.

[8] Corritore, C. L. and Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50(1), 61-83.

[9] Cox, B. (1986). *Object-Oriented Programming*. Reading, MA: Addison-Wesley.

[10] Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1996). Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1, 109-132.

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

[12] Gwinn, J. (1992). Object-oriented programs in realtime. *SIGPLAN Notices*, 27(2), 47-56.

[13] Henry, S. and Humphrey, M. (1993). Object-oriented vs. procedural programming languages: effectiveness in program maintenance. *Journal of Object-Oriented Programming*, 6(3), 41-49.

[14] Jacky, J. and Kalet, I. (1987). An object-oriented programming discipline for standard Pascal. *Communications of the ACM*, 30(9), 772-776.

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

[16] Letovsky, S. (1986). Cognitive processes in program comprehension. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers* (pp. 80-98). Norwood, NJ: Ablex.

[17] 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.

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

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

[20] Rajlich, V., Doran, J., and Gudla, R. T. S. (1994). Layered explanations of software: a methodology for program comprehension. *Proceedings of the Third Workshop on Program Comprehension* (pp. 46-52). Los Alamitos, CA: IEEE Computer Society.

[21] Rentsch, T. (1982). Object-oriented programming. *SIGPLAN Notices*, 17(9), 51-57.

[22] Shaft, T. M. and Vessey, I. (1995) The relevance of application domain knowledge: the case of computer program comprehension. *Information Systems Research*, 6(3), 286-299.

[23] Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, 8, 219-238.

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

[25] 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.

[26] Tilley, S. R., Paul, S., and Smith, D. R. (1996). Towards a framework for program understanding. *Proceedings of the Fourth Workshop on Program Comprehension*, (pp. 19-28). Los Alamitos, CA: IEEE Computer Society.

[27] Von Mayrhauser, A. and Vans, A. M. (1995a). Program comprehension during software maintenance and evolution. *Computer*, 28(8), 44-55.

[28] Von Mayrhauser, A. and Vans, A. M. (1995b). Program understanding: models and experiments. In M. C. Yovits and M. V. Zelkowitz (Eds.), *Advances in Computers*, vol. 40, (pp. 1-38). San Diego: Academic Press.

[29] von Mayrhauser, A. and Vans, A. M. (1996). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6), 424-437.

[30] von Mayrhauser, A. and Vans, A. M. (1997). Program understanding behavior during debugging of large scale software. In S. Wiedenbeck and J. C. Scholtz (Eds.), *Empirical Studies of Programmers: Seventh Workshop* (pp. 157-179). Norwood, NJ: Ablex.

[31] von Mayrhauser, A. and Vans, A. M. (1998). Program understanding behavior during adaptation of large scale software. *IWPC'98: Proceedings of the Sixth International Workshop on Program Comprehension* (pp. 164-172). Los Alamitos, CA: IEEE Computer Society.