EXPERT PROBLEM SOLVING STRATEGIES FOR PROGRAM COMPREHENSION

Jürgen Koenemann and *Scott P. Robertson*

Department of Psychology
Rutgers – The State University of New Jersey
New Brunswick, NJ 08903
koeneman@paul.rutgers.edu

ABSTRACT

Program comprehension is a complex problem solving process. We report on an experiment that studies expert programmers' comprehension behavior in the context of modifying a complex PASCAL program. Our data suggests that program comprehension is best understood as a goal–oriented, hypotheses–driven problem–solving process. Programmers follow a pragmatic as-needed rather than a systematic strategy, they restrict their understanding to those parts of a program they find relevant for a given task, and they use bottom-up comprehension only for directly relevant code and in cases of missing, insufficient, or failing hypotheses. These findings have important consequences for the design of cognitively adequate computer–aided software engineering tools.

KEYWORDS

Software Psychology, Program Comprehension, Protocol Analysis

INTRODUCTION

> Yes, code please ... I'll probably go to the comments after that, when I realize that I don't understand the code. [Subject 08]

A central activity of the software maintenance process is *program comprehension*, the process of understanding program code unfamiliar to the programmer. The goal of our research is to provide strong empirical support for the paradigm that program comprehension is a hypotheses–driven problem–solving process. Consequently, cognitively adequate user interfaces for computer–aided software engineering tools have to support these empirically identified processes.

There is a growing body of research on the issue of program comprehension. Cognitive research since the early 1980's has focussed on comprehension strategies, the procedural aspect of expertise, as well as on the declarative aspect of how programming knowledge in general and the program under consideration in particular are conceptually represented [1, 5, 15, 19, 23, 25, 26].

A study by Littman et.al.[17] (see also [16]) identified two equally dominant comprehension strategies: Programmers used either a *systematic* or an *as–needed* strategy. The former group used extensive symbolic execution of the data and control flow between subroutines to gain a detailed understanding of the program *prior* to modifying the code, whereas the latter group tried to minimize the understanding of the program by localizing those parts of the program that needed to be changed and studying those parts only. This view of program comprehension as a hypotheses–driven problem–solving process was first put forward by Brooks [3] in his model of "Beacons" that guide comprehension. Other studies [27, 28] demonstrated the role of code level beacons for understanding small, simple programs. Research at MCC on initial stages of software design ([8, 9, 10, 11, 14]) and comprehension [2, 4] also demonstrated the heuristic approach to software engineering tasks.

In contrast, there is a body of research that views program listings as a special class of display of information [7] and, consequently, sees program comprehension to be similar to (prose) text comprehension. Parsing models of the text comprehension literature [13] are adapted for the use with programming languages to predict reading times based on the "micro–structure" of program statements [6, 18]. Comprehension, understanding of the "macrostructure" of program code, is seen as a bottom–up chunking process [20, 21, 24]. Robertson et.al. [22] show that complete knowledge of the micro– and macro–structure of a program is not sufficient to predict the comprehension behavior of programmers. Here we show that subjects follow a pragmatic "as–needed" strategy rather than a systematic approach, that subjects restrict their understanding to parts of the code they consider to be relevant for the task and, thus, gain only a partial understanding of the program that might lead to misconceptions or errors. Tools will have to be developed that facilitate "as-needed" strategies and help programmers to avoid some of its inherent problems.

METHOD

Subjects

Twelve subjects participated in the study and were assigned randomly to the four different tasks. Subjects knew at least three programming languages, had programmed for 4 to 15 years (avg. 7.6), had PASCAL experience for at least 4 years, worked with large programs on a routine basis, and had a formal education in computer science or electrical engineering.

Task	Avg. # of Pieces of Information Studied: Unique Visits C. (43)	D. (40)	Ch. (6)	All Visits C.	D.	Ch.
A	10.0	2.7	2.7	12.7	3.3	5.0
B	11.3	4.0	1.3	23.3	4.3	1.3
C	7.0	5.3	3.0	13.0	5.7	3.7
D	6.7	3.0	2.0	8.0	3.0	3.0
Avg.	8.7	3.8	2.3	14.3	4.1	3.3
C.=Code Listings, D.=Descriptions, Ch.=Charts						

Table 1: Usage of Information Sources in Program Comprehension Process

Subjects were paid $ 10 per hour for their participation.

Material

The computer program to be modified was a troff-style text formatting program. The 636 lines long, uncommented, well structured PASCAL program was adapted from a standard text on software engineering [12]. It consisted of a short main program and 39 functions and procedures. For each procedure or function there existed a code listing and a separate description of its functionality. In addition, six module decomposition charts were provided. The program–related material was complemented by a set of three global program descriptions.

There were four modification task descriptions, each outlining the changes to be made on a functional level. Tasks varied in difficulty, required level and scope of program understanding, and amount of code to be added or changed. Tasks included a functional addition (boldfacing; task A), enhancement (footer/header centering; task B), a functionality change (underlining; task C), and a default value change (default footers; task D).

Procedure

Subjects read a summary of the program to be modified and descriptions of all four modification tasks. Subjects then verbalized how they would perform each task. Next, subjects were given one modification and they studied program code or the documentation units, one piece at any given time, until they completed the modification. During this phase subjects were requested to think aloud and to verbalize and explain their requests for a piece of information. This phase was audiotaped and the experimenter recorded the sequence of information requests. Subjects were advised to do the modification in an efficient, structured way.

RESULTS AND DISCUSSION

General Analysis

Subjects took between 15 and 44 minutes to complete the modification phase of the experiment ($\overline{x} = 28.25$, $sd = 7.3$). Modification time was correlated with the number of pieces of information looked at ($r(10) = .73, p < .01$).

table 1 details which information sources were used by the subjects most frequently. Subjects used the code listings as their main source of information. Subjects did use the module decomposition charts but rarely made use of the detailed descriptions of each code segment. The comparison of unique versus all visits reveals that subjects looked on average only about 1.5 times at any particular code segment

Task	Code Studied: % Parts (of 43)	% Lines (of 635)	Existence of Code **not** known: % Parts (of 43)	% Lines (of 635)
A	23.3	36.2	18.6	14.7
B	26.4	35.4	24.8	21.8
C	16.3	19.8	19.4	17.5
D	15.5	21.7	33.3	31.2
Avg.	20.4	28.3	24.0	21.3

Table 2: Scope of Comprehension Process

or chart and only once at any piece of accompanying documentation. This suggests that subjects retrieved the desired information the first time they looked at it.

The very limited overall scope of comprehension is detailed in Table 2. Subjects studied only a small part of the program code; on average only one fifth of the procedures and functions, accounting for less than one third of all lines of code in the program, was visited. None of the subjects studied more than about half of the code and one subject restricted himself to one tenth of the code. Furthermore, Table 2 shows the percentage of procedures and functions that subjects were totally unaware of since they did not even encounter the corresponding calls or references in charts or descriptions. On average, 24% of all functions or procedures (accounting for 21.3% of the code) were totally missed. Note, that this is only a lower bound since subjects often glanced over information and were, thus, likely to miss existing references to other code segments as well.

We claim that the small amount of code studied results from an opportunistic *relevance strategy*: Subjects study code or documentation only if they believe that the code is relevant for the task. We can separate three different levels of relevance: *Directly* relevant are those code segments that have to be modified or that are used for copy–editing purposes. This code is the common core for all subjects that solve the same tasks. Since this direct relevance is task dependent, subjects with different tasks study different parts of the program. For example, the underlining procedure `underln` was studied by all subjects with boldfacing/underlining tasks, but by none of the other subjects, whereas `puttl`, the procedure that put out footers and headers, was studied by 5 out of 6 subjects performing a footer related modification, but by none of the other subjects.

The second category is *intermediate* relevant code. This group includes those code segments that are perceived to interact with relevant code. Subjects study intermediate relevant code if they discover the interaction and if they judge that the interaction needs to be understood. Intermediate relevant code is studied less frequently and in less detail than directly relevant code.

The third group of code (and documentation) is of *strategic* relevance in that it guides the comprehension process. This code is rarely comprehended in detail but serves mainly to locate or detect directly or intermediate relevant code, that is, it acts as a pointer towards other relevant parts of the code. Prototypical examples are charts which were used by subjects to locate procedure names and documentation that was used to determine the calling procedures for a given code segment; e.g. all subjects studied the code or chart of `main` and the procedure `command`.

We also found that a quarter of all procedures and functions were not looked at by *any* subject. Most of these were low

level procedures like `max`, `min`, or `isdigit`, suggesting that subjects had common knowledge about the functionality of these procedures and, thus, saw no necessity to look at the code in detail or judged them as irrelevant. However, some of these code segments were indeed complex and important domain–specific procedures.

These data qualifies the finding that a systematic strategy plays an important role for comprehension [17]. Our data show that subjects did not employ a systematic comprehension strategy but focussed on a small subset of the code. This finding gives strong support to the dominance of "as-needed" strategies, "need" being predominantly determined by the modification goals of the programmer. We attribute these differences to the fact that our program was significantly larger in size. With increasing program size and complexity, any comprehensive strategy will become cost prohibitive and systematic in-depth comprehension will be restricted to few relevant code segments. In agreement with Letovsky and Soloway [16] we found that subjects employing the "as–needed" strategy performed generally well but failed to notice some delocalized interactions. However, most of the missed interactions would have become immediately evident if subjects were given the opportunity to test their modifications online.

Differences in relevance make some types of information more important during particular phases in the comprehension process. Figure 1 shows the varying usage of information over the course of the comprehension process. Each subject's process is broken down into quartiles. The vertical axis shows the distribution of types of information studied. The percentages given are averages for all subjects. As expected, information of mainly strategic relevance, namely charts and the global descriptions, was of importance at the beginning of the comprehension process. Global descriptions were either looked at first or not at all. The usage of charts decreased linearly over time, $F(1, 11) = 36.05, p < .001$. Conversely, code use increases linearly over the span of the comprehension process, $F(1, 11) = 33.00, p < .001$. Descriptions of code segments fluctuate and their use reached the peak in the third quartile. This may be due to the usage of descriptions as pointers to the call structure of the program (strategic relevance)in the beginning and in addition for clarification in cases where subjects failed to understand the code itself.

One might argue that our findings are an effect of the procedure of manually presenting one piece of information at any given time and on request only. Whereas we agree that the chosen method constitutes some deviation from a realistic setting we would like to point out that all subjects reported after the experiment that they felt they had performed the modification in their "normal" way and that they were not hampered by the chosen method. Furthermore, the procedure was quite similar to a software environment in which subjects view one code segment at any given moment and switch between segments using editor commands that refer to code segments by name.

Detailed Analysis of One Modification Task

We will now focus on one particular task, the automatic centering of headers and footers (Task C). This modification required the insertion of code into the procedure `puttl` or the insertion of identical code in the two calling procedures `puthead` and `putfoot`. Figure 2 depicts the comprehension process for each of the three subjects solving this task. Groups of code segments and documentation are listed on the vertical axis in a partial order that reflects the call structure of

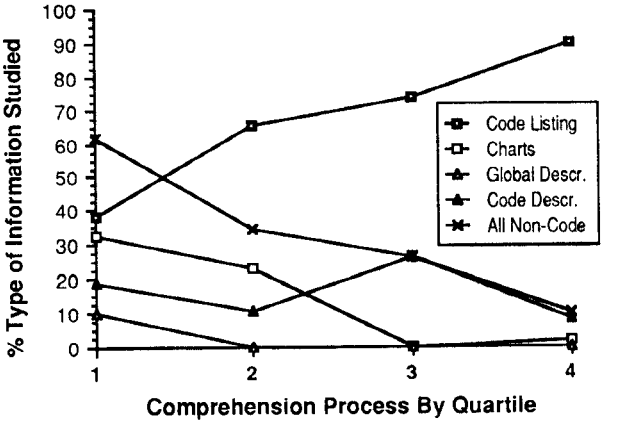

Figure 1: Shifts in Information Usage During Program Comprehension

the program. The horizontal axis represents time. The shape of the symbols depicts the information type.

Subjects uniformly exhibited a focussed top–down process. They first got an overview of the program by studying the global descriptions and proceeded with the goal to search for the directly relevant code that printed footers and headers.

> One thing I want to do a little bit is hop down looking at the program. But I think I'll stop in main, just to get a, an idea of which path I should take next. [Subject 13]

> I'd like to just have an overview of the capabilities of the program, so that I understand what I'm getting into. I mean, I-I, I assume that, that my employer wants me to do it right and not fast. If he wants me to do it fast, I could ask you maybe for the module chart, guess where headers and footers are done, and try to dive right in. But I feel more comfortable to have an over–overview of program functionality, even though most of this information I won't use directly.[Subject 07]

How did subjects decide whether a particular code segment was of direct, intermediate, or strategic relevance? Subjects hypothesized relevance based on the comprehension of the program already achieved and based on their knowledge about the task domain and programming in general. Subjects used procedure and variable names to infer functionality. The names chosen for the formatter program consisted of mostly abbreviated or condensed function descriptions, e.g. `getcmd` for `getcommand`. Subjects in all tasks demonstrated great ability to decode these names.

> Define footer title. Uh, th–that explains the, the name of `puttl` and the use of the word title everywhere. Cause I guess using title is a sub – a special line. [Subject 10]

> Um. `text. leadbl` – lead balloon, who knows what that is? – leading blanks, OK, there we go. [Subject 08]

Furthermore, subjects expected consistency in the use of names. New variables or procedures introduced by the subjects were named following the given naming format. While looking for code subjects guessed correctly the names of procedures they had not seen.

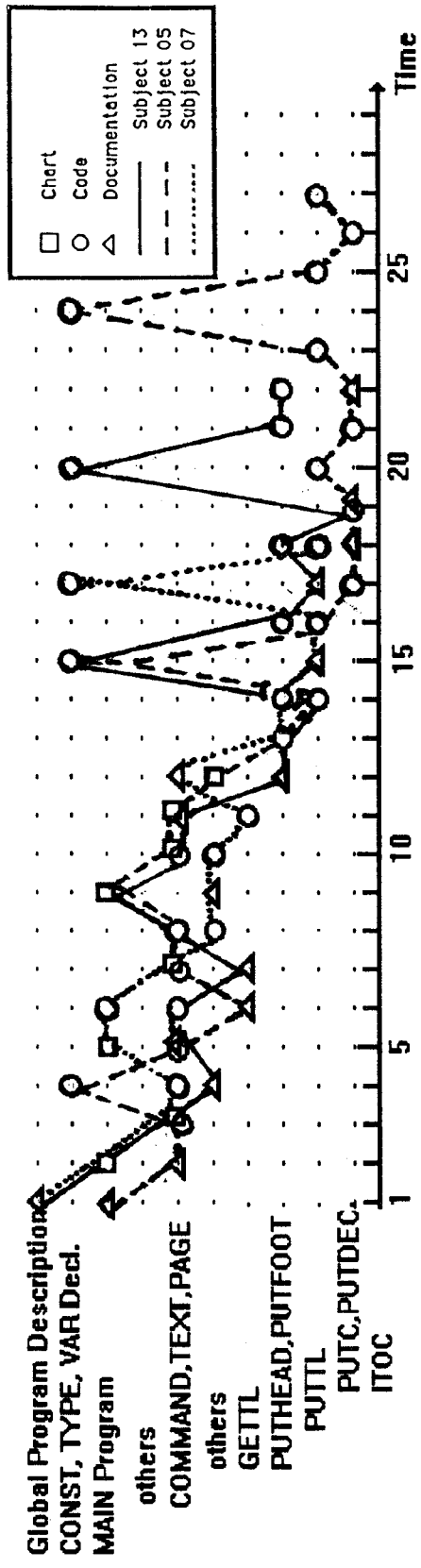

Figure 2: Comprehension Process for Solving Task C

> Can I make a guess that there's a, that there's a companion routine called `puthead`. Even though I'm not sure where it's called from? And I don't really care? [Subject 13]

> So let me look at `putfoot` first. [...] I think it puts out the page footer. And I'm also guessing. I haven't seen anything called `puthead` yet. [...] You know, cause you like to believe other programmers are rational and , uh–Oh, it's not gonna – It looks like it may not matter anyway because they're both done by separate team called `puttl`. I don't know they're both done by `puttl`, but on line 350 it says `puttl` footer current page. And I would guess if I look at `puthead`, I'll see `puttl` header current page. [Subject 07]

After subjects failed to find directly relevant code, subjects backed up to the calling routines and/or charts. Subjects then hypothesized that the procedure `page` was printing out a page footer and was starting a new page since that is the use of a page command from the user's perspective (steps 8-12):

> I'm guessing that `page` is the function which handles that page feed which is where we're gonna have to hook a header into. [Subject 05]

> I think it's gonna be somewhere under `page`, um, where it generates the header and footer. [Subject 07]

> ...let me look at the description for the code that's called `page`. I'm now assuming that the header and footer strings have got read into those variables, and are gonna be used by the `page` function.[Subject 13]

All subjects reached the code listing for the `putfoot` procedure in step 13. They than either studied the parallel procedure `puthead` or proceeded to the code for `puttl` that had to be modified. Subjects began the coding of the modification as soon as they reached directly relevant code. The length of this last step in the comprehension process varied. All subjects looked up at least one value in the global declarations. The extra steps by Subject 05 resulted from a copy–edit episode, a common strategy employed by many subjects.

> Can I see the description on `itoc`? Because it looks like that's what's doing the the, the actual conversion of the number. And maybe I can see how they are finding–checking the width of that number. You know, maybe I can borrow their code...[Subject 05]

Throughout, subjects made relatively little use of the provided documentation. The protocols suggest documentation was mainly used in the case of breakdowns in the comprehension process and for strategic purposes, namely the identification of calling procedures.

> Yes, code please [...] I'll probably go to the comments after that, when I realize that I don't understand the code. [Subject 08]

> I'll look at the description first. [laughs] I'm getting a little bit more, um, conservative here ...

> I might actually have to look at some documentation soon ... Did I miss – ? Must have missed the call [...] I seem to have lost the command parser. I'm gonna have to look at the documentation on , uh, `main` and `getline`. [Subject 06]

> Where did that come from? Must be missing something when looking at the `command` code, so let's take a look at the functional description of `command` ...[Subject 02]

The only documentation frequently used were charts due to their strategic relevance. The abstraction to procedure and function names and their call structure supported the subjects' strategy of finding particular procedures by searching for their names. Thus, charts were looked at often early on, but once the relevant code segments had been located (after step 13), charts were no longer used.

CONCLUSION

The modification of a large program is a complex problem–solving task. We discussed how subjects generate hypotheses about the functionality of code and how procedure and variable names are used as "beacons" to build new, revised, or refined hypotheses during the comprehension process. We were, thus, able to demonstrate that program comprehension proceeds mainly in a top–down fashion and that bottom–up methods of comprehension, the integration of single code units into meaningful frames, were only used in cases of missing or failing hypotheses and locally for directly relevant code units. We found only two instances across all four tasks in which subjects actually simulated intensively the execution of the code in order to understand a code segment.

No subject followed a systematic strategy of comprehension. The goal of modifying the program according to the task at hand was the major factor that determined the scope of the comprehension process. Subjects spent a major part of their time searching for code segments relevant to the modification task and no time understanding parts of the program that were perceived to be of little or no relevance.

These findings suggest that tools like the DESIRE system [2] that are based on a paradigm of programming as a design task, utilize the reuse of existing code and allow the expression and refinement of and reasoning about abstract concepts and constraints, are the right approach towards the development of an "intelligent" programmer assistant.

Subjects reported that they were constrained by the fact that they could only view one piece of information at any given time and that switching between two pieces of information had a high "cost". This suggests that multi–window environments that allow programmers to view different information sources simultaneously and large windows that facilitate speedy scanning of large portions of code are highly desirable.

The extensive browsing behavior of subjects and the scanning of charts and code for relevant procedure names should be supported by graphical browsers and by intelligent search functions, utilizing online documentation, that allow the programmer to use functional categories in queries to retrieve procedure names.

A second area with large potential for support is the process of reusing existing code that has to be copy–edited. For example, a system should support the declaration and initialization of variables. A tool that would allow the user to establish the correspondence between a new piece of code and an existing piece of code (e.g. "bfval" is like "ulval") and that find other occurrences of that code segment and ask the user whether analogous code should be added for those occurrences as well would be helpful in supporting reuse and might reduce the numbers of errors made.

Documentation was mainly seen as the last resort and only consulted, with the exception of flow–charts, when other methods of comprehension failed. Many subjects reported that they had bad experiences with useless documentation and even a single instance of misleading code description during the experiment caused one subject to neglect descriptions thereafter. Subjects try to avoid the extra effort of studying documentation if they believe that the information can be obtained directly from the code. On the other hand, subjects do use documentation when they know that code itself does not provide the desired information, e.g., in situations where a subject wants to find all calling procedures for a called subroutine. These findings suggest that documentation should be tailored to include only those pieces of information that are not directly obtainable from the code, e.g. the names of calling procedures. To minimize the cost of looking at documentation, it should be readily available. This favors the inclusion of documentation in the code itself via comments and the selection of self-explanatory variable and procedure names. Multi-window environments that allow the parallel display of code and documentation might also encourage the use of documentation. The core issue is why much documentation is perceived as useless. The (modification) task on hand determines the scope and focus of comprehension. For one modification it might be sufficient to know how a piece of code works while for a different modification the question of why this implementation was chosen is of great importance. If program comprehension is understood as design reconstruction, documentation should facilitate this process by revealing parts of the original design *process* that cannot be easily reconstructed from the resulting code. For example, it might be documented why a particular encoding has been chosen from a set of alternatives. This perspective gives rise to two types of documentation: *design history* documentation, allowing a programmer to examine the decision processes that resulted in the original design and *anticipatory documentation*; documentation is created explicitly to support (likely) future modifications.

ACKNOWLEDGEMENTS

This work is sponsored in part by the Office of Naval Research, Perceptual Science Program Contract N00014–86–K–0876. We thank the three anonymous reviewers for their valuable comments and Chris Jarocha-Ernst for helping with the preparation of the document.

REFERENCES

1. B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9:422–433, 1981.
2. Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–47, 1989.
3. Ruven E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
4. B. Curtis, S.B. Sheppard, J. Kruesi-Bailey, and D. Boehm-Davis. Experimental evaluation of software

specification formats. *Journal of Systems and Software*, 9(2):167–207, 1990.

5. Francoise Detienne and Elliot Soloway. An empirically–derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, 33(3):323–342, September 1990.
6. Jennifer Dyck and Brent Auernheimer. Comprehension of pascal statements by novices and expert programmers. Poster presented at the Human Factors in Computing Systems CHI-89 Conference, Austin, TX, 1989.
7. D. J. Gilmore and T. R. Green. Comprehension and recall of miniture programs. *International Journal of Man–Machine Studies*, 21:31–48, 1984.
8. Raymonde Guindon. Software design tasks as ill–structured problems, software design as an opportunistic process. Technical Report STP-214-88, Microelectronics and Computer Technology Corporation, 1988.
9. Raymonde Guindon. The process of knowledge discovery in system design. Technical Report STP-166-89, Microelectronics and Computer Technology Corporation, 1989.
10. Raymonde Guindon. What knowledge is exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3):279–304, September 1990.
11. Raymonde Guindon, Herbert Krasner, and Bill Curtis. Breakdown and processing during the early activities of software design by professionals. In Gerry Olsen, Eliot Soloway, and Sylvia Sheppard, editors, *Empirical Studies of Programmers. Second Workshop*. Ablex Publishing, 1987.
12. Brian W. Kernighan and P. J. Plauger. *Programming Tools in Pascal*. Addison-Wesley, Reading, MA, 1981.
13. Walter Kintsch. *The Representation of Meaning in Memory*. Erlbaum, Hillsdale, NJ, 1974.
14. Herb Krasner, Bill Curtis, and Neil Iscoe. Communication breakdowns and boundary spanning activities on large programming projects. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers. Second Workshop*, pages 47–64, Norwood, NJ, 1987. Ablex Publishing.
15. Stanley Letovsky. Cognitive processes in program comprehension. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers. First Workshop*, pages 58–79, Norwood, NJ, 1986. Ablex Publishing.
16. Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.
17. David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers. First Workshop*, pages 80–98, Norwood, NJ, 1986. Ablex Publishing.
18. Richard E. Mayer. A psychology of learning basic. *Communications of the ACM*, 22:589–594, 1979.
19. G. M. Olson, Sylvia Sheppard, and Elliot Soloway, editors. *Empirical Studies of Programmers: Second Workshop*, Norwood, NJ, 1987. Ablex Publishing.
20. Nancy Pennington. Comprehension strategies in programming. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers. Second Workshop*, pages 100–113, Norwood, NJ, 1987. Ablex Publishing.
21. Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
22. Scott P. Robertson, Erle F. Davis, Kyoto Okabe, and D. Fitz-Randolf. Program comprehension beyond the line. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Proceedings of the 3rd International Conference on Human–Computer–Interaction, Interact–90*, pages 959–970. Elsevier Publ., 1990.
23. Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Cambridge, 1980.
24. Ben Shneiderman and Richard E. Mayer. Syntactic/semantic interactions in programmer behavior. a model and some experimental results. *International Journal of Computer and Information Sciences*, 8:219–238, 1979.
25. Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.
26. Elliot Soloway and Sitharama Iyengar, editors. *Empirical Studies of Programmers*, Norwood, NJ, 1986. Ablex Publishing.
27. Susan Wiedenbeck. Processes in computer program comprehension. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers. First Workshop*, pages 48–57, Norwood, NJ, 1986. Ablex Publishing.
28. Susan Wiedenbeck and Jean Scholtz. Beacons and initial program comprehension. Poster presented at CHI Human Factors in Computer Systems, 1989.