# Software Psychology Requirements for Software Maintenance Activities

*Spencer Rugaber*

*Victoria G. Tisdale*

Software Engineering Research Center
Georgia Institute of Technology

## 1. Introduction

Software psychology attempts to discover and describe human limitations in interacting with computers. These limitations can place restrictions on and form requirements for computing systems intended for human interaction. Hypermaint is such a system. Hypermaint is designed, with human limitations in mind, to facilitate the human task of maintaining software.

Software maintenance encompases all activities performed on a piece of software intending to keep it useful. This includes efforts to keep software at the same level of performance/usability, as well as efforts to improve it. In the process of software maintenance, source code of the software must be examined and understood by the maintainer.

The task of maintenance can be facilitated by a system congruent with human abilities and limitations. Experimental evidence of these abilities and limitations are discussed in Shneiderman's "Software Psychology : Human Factors in Computer and Information Systems".

"Software psychology is a study of human performance in using computer and information systems",[9] Software psychology incorporates the methods and knowledge of multiple fields within the area of psychology. It applies the techniques of experimental psychology to analyze human performance aspects in computer tasks. It also applies the concepts of cognitive psychology to the cognitive and perceptual processes involved in computer interaction. The methods of social, personnel and industrial psychology aid the understanding of human decision making ability, congnitive style, and personalities of those working in the computer science field. The theories of psycholinguistics are explored in hopes of greater understanding of human - computer language performance.

"The goal of software psychology is to facilitate the human use of computers",[9] Human characteristics in the work environment have to be analyzed. In the case of maintenance, the understanding of human skills and capacity to work with software is necessary in order to facilitate the maintainer's examination and understanding of source code. Strengths and limitations of human abilities serve as underlying factors in determining the functionality of software maintenance tools. An understanding of software psychology will be used to aid in the design of the underlying information structures of a software maintenance system and in the design of the human-computer interface of this system to enable better and more efficient understanding of source code.

In studying human performance in human-computer interaction, a number of areas need to be addressed. First, some specific programming activities are examined. Second, cognitive structures and processes that are involved in specific human-computer interactions are proposed and tested by memorization/reconstruction and other experiments. Last, some human factors issues of programming activities are discussed.

## 2. Areas Of Programming Activity

Software psychologists focus on such human factors as: ease of use, simplicity in learning, improved reliability, reduced error frequency, and enhanced user satisfaction. Particular areas of

programming activity about which experiments have been performed are program comprehension, composition, debugging, and modification. Different types of experiments are performed in each of these areas.

### 2.1. Program Comprehension

Of most relevance to software maintenance is program comprehension. Program comprehension is studied via memorization/reconstruction experiments of programs. These experiments have provided valuable insight into the cognitive structures and processes involved in various programming tasks.

A memorization/reconstruction task consists of studying a program and then reconstructing it from memory. Experience of the subjects plays a vital role in this task. An experiment by Shneiderman[11] showed that as experience increased, the ability to reconstruct the proper program increased rapidly. Often experienced programmers wrote functionally equivalent, but syntactically different versions. His hypothesis is that as subjects gain experience in programming they improve their capacity for recognizing meaningful program structures, thus enabling them to recode the syntax into a higher-level internal semantic structure.

Shneiderman suggests that "...performance on reconstruction task is a good measure of program comprehension. Memorizing complex material such as a computer program can not be accomplished by rote memorization, but knowledgeable programmer can successively recode program statement groups into ever higher-level semantic structures." Success at reconstruction indicates that the subject understands the low-level details of each statement, intermediate-level groupings, and the overall function of the program. This seems to be the best definition of program comprehension: "...the recognition of the overall program function, an understanding of intermediate-level processes including program organization, and comprehension of the purpose of each program statement."

The work in memorization of programs closely parallels the experiments done in the area of memorization of English sentences. The experiments with memorization of English sentences [1, 2] show that the syntactic information is retained only briefly, until it is converted to semantic information, and then lost. Semantic information, however, is stored in long-term memory and can later be recovered, but possibly in a different syntactic form.

After the subject completes the reconstruction task, his reconstructed code is analyzed in terms of information chuncks which give insight to the kinds of internal information structures contained the code. It is in terms of these internal structures that the code is interpreted, or understood, by the subjects. Shneiderman identifies these stuctures in his model. ( see section "Models of Program Comprehension and Problem Solving" )

Program comprehension could be viewed as one type of a problem solving task. The goal of program comprehension is to understand the given code. Greeno proposes a model of problem solving and describes the processes and mental structures involved in problem solving. His model could be applied to program comprehension. ( see section Models of Program Comprehension and Problem Solving )

### 2.2. Program Composition

Program composition is studied directly by asking the subjects to write programs according to some specifications. Grading of programs is done to predesigned standards to ensure consistency.

### 2.3. Program Debugging

Debugging involves locating and correcting errors in a program. These errors are of two types : syntactic and semantic. Syntactic errors consist of incorrect syntax of the programming language. Most of these errors are found by the compiler and do not pose a serious problem. Semantic errors are errors in design or composition. Semantic errors are more difficult to find if they are not obvious from the output of the program. Design errors are difficult to find because the programmer must see his code as he coded it, not as he meant to code it.

Subjects in debugging experiments are required to locate and correct the bugs in a given program. Perforamance is evaluated by a count of found and corrected bugs. Points can be deducted for finding nonexistant bugs and added for finding bugs unknown to the experimenter.

### 2.4. Program Modification

Program modifications consume 25 to 75 percent of all programming efforts.[9] It requires competence in program comprehension and composition. Programmers are required to understand the program in a short time, and make changes to it that do not interfere with the rest of the program's execution. Subjects of modification experiments are given instructions about the modifications they are to perform. Performance is evaluated on the number of successful modifications.

## 3. Models of Program Comprehension and Problem Solving

The majority of the program maintenance tasks are founded on program comprehension. Shneiderman's model identifies the main cognitive structures involved in program comprehension. He also distinguishes two types of information with the respective cognitive structures which process them. Greeno's model is a general model of problem solving which can be applied to program comprehension. This model describes the processes and mental structures involved in problem solving.

### 3.1. Shneiderman's Model of Program Comprehension

Shneiderman views the comprehension of programs as consisting of three levels: low-level comprehension of the function of each line of code, mid-level comprehension of the structure of the algorithm and data, and high-level comprehension of the overall program function. It is possible to understand each line of code and not to understand the overall program function. It is also possible to understand the overall function of the program yet not understand the individual lines of code, nor the structure of the algorithms and data. The mid-level comprehension involves knowledge of the control structures, module design, and data structures, which can be understood without knowledge at the other two levels. Thorough comprehension involves all three levels of understanding.

An experienced programmer possesses a network of multi-leveled knowledge in his long-term memory. Some of these multi-level concepts, which are extracted from experience and independent of programming language or environment, comprise the programmer's semantic knowledge. Semantic knowledge consists of multi-level concepts such as the knowledge of what an assignment statement does, how a stack is implemented and how it can be used, strategies for sorting a set of elements, and many others.

Another kind of information stored in programmer's long-term memory is syntactic knowledge. This knowledge consists of syntactic details of different programming languages and systems such as: the proper positioning of semicolons, the proper symbols for assignment and conditional statements, available data types, and other features of the language or environment.

Shneiderman's views comprehension as a process of converting the code of a given program to some internal semantic form. This conversion is achieved with the help of the programmer's semantic and syntactic knowledge. At the highest level, the programmer forms an idea of the program's purpose. He then recognizes lower-level structures such as algorithms for sorting, searching, familiar streams of statements, or others. Finally, he reaches an understanding of what the program does as well as how it does it. This understanding is represented in some internal form. This internal representation of the program is independent of the syntactic form from which it was extracted and is capable of being expressed in other languages or environments.

### 3.2. Greeno's Mental Model of Problem Solving

Software psychologists are also working to uncover and describe the underlying mental structures and processes involved in various programming activities. Since some activities involved in maintenance are comparable to problem solving activities, models of problem solving could be transferable to these activities.

A model proposed by Greeno[4] looks at the components of memory involved in problem solving tasks. In this model, information about the problem to be solved is presented to the programmer and placed in his short term memory. His past knowledge about the problem is retreived from long term memory and placed in working memory. New information is then transferred to working memory where it is integrated with the information from long term memory. The result of this integration of

information is a solution to the problem or learning of the new information, which is the integration and placement of new information into long term memory.

The transfer of the problem description from short term memory to working memory, and the recall of appropriate knowledge from long term memory to working memory, represent the first phase of problem solving. [14] During the first phase, working memory's contents are prepared to solve the problem. During the second phase, the programmer's solution plan emerges in the form of internal semantics, an internal representation of the solution. Once the internal semantics of the problem are constructed, derivation of the solution is straightforward.

The internal semantic representation starts as a general sketch of a solution and develops into a more detailed plan with time. The detailed plan consists of a number of subplans, parts of the total problem and specific solutions to them. This is the underlying concept of modularization, subdividing the problem into its composite problems and solving them.

Defining subgoals for the programmer facilitates the development of the internal semantics of the problem[3, 8] Providing the programmer with a general definition of the program would provide a starting point. It would also be helpful to subdivide the program into parts whose interaction is minimal. This would suggest some level of modularity. In construction of the internal model of the semantics of the program, it would be helpful to have a medium for representing, documenting, and storing the internal model as it is being developed. This model, of course, varies highly between programmers. Nevertheless, some tailorable high-level medium could be helpful.

### 3.3. Closure

Short term memory has limited capacity. In order to retain information in short term memory, it must be constantly rehearsed, which requires effort. Humans experience a feeling of relief when information no longer needs to be rehearsed and can be forgotten. This relief, experienced at the completion of a task, is termed closure. The need for closure suggests that it would be preferable to work with small portions of a task at a time and to be able to release the information at completion of that portion.

Understanding of code could be easier if the programmers worked on understanding of smaller semantic segments of code, or modules of code. Unfortunately, working with undocumented poorly designed code, the programmer is forced to read the code in its entirety to locate possible semantic modules. However, once these modules are identified, the maintenance task could be facilitated by allowing the user to view semantic clusters of code at a time.

It is also useful to help the programmer in identifying semantic modules. This could be achieved by supporting possibly related user queries, such as letting the user find the piece of code where a certain variable is modified, where it is being declared, where a certain data structure is changed or updated.

## 4. Viewing/Style Issues

A number of factors play a role in the transformation of code to an internal semantic form. These factors deal with the viewing and the style of the program code. Experiments have been performed to determine the extent to which these factors affect the comprehension of programs.

### 4.1. Commenting

The issue of the influence of comments on program understanding is not resolved. A number of studies of short programs[7] show that comments in the code interfere with the process of understanding, require more filtering when reading the code, and if not updated, could be misleading and cause errors in the semantic representation of the code. Comments in the code make programs longer and disrupt the flow of code. Experiments with longer, more realistic, programs are not reported by Shneiderman. It may well be the case that the importance of comments increases with the length of the program.

Some experiments sited by Shneiderman [12, 13] show that functionally descriptive comments do facilitate faster conversion of code to internal semantic structure, while non-descriptive comments hinder it. Functionally descriptive comments used in these experiments are high-level comments that

describe actions or effects not obvious from viewing the code. Low-level non-descriptive comments which restate the function of the code hinder program understanding by unnecessarily interrupting the subject's thought process.

Most programmers still prefer to use some comments. For this reason it is useful to allow a link between a comment and a part of code that it belongs to. This link would give an option of removing the comments from the code, or overlaying them on the code, for both kinds of viewing.

### 4.2. Variable Names

The use of mnemonic variable names seems to be helpful in program comprehension.[6] The mnemonics, however, have to be such that they add semantic information relevant to the code. It is most likely that different mnemonics have different meanings to different programmers. Allowing for systematic substitution of variable names according to programmer's specifications could tailor the code to his personal preference. Having meaningful mnemonics would releave the programmer's STM load, making his task easier.

### 4.3. Indentation

Most programmers use indentation, but experimentally, the advantages of indentation have not been substantiated. Experiments by Weissman show that indented and commented programs are more difficult to read. Love[5] shows that indentation does not improve understanding for short FORTRAN programs, and Shneiderman and McKay [10] show that indented long programs are more difficult to read because deep indentation can cause lines to split to accomodate margins. Since there is great variability in display preferences which could change the size and indentation look of the code, and variability in indentation preferences, it is useful to allow the programmer to specify his indentation preference which would uniformly apply to the code.

## 5. Conclusion

From the above presented discussions, a number of implications can be seen (Appendix A). These rest on human limitations and facilitate the task of understanding code.

In comprehending a program, the first task is to represent a given program in some internal semantic form. This begins with an understanding and a formulation of a general idea of what the program does. Providing the programmer with a general definition of the program is helpful in this first step.

In the maintenance task, the modification or enhancement to be made must also be understood and encoded into an internal semantic form. Once both the internal semantics of the change and of the program are formed, the internal semantics of the program are modified in accordance with the semantics of the change. This encoding accomplished, the foundation for implementation of the code is set, and the programmer is ready to implement the change. It would be helpful to have a medium for representing, documenting, and storing the internal model as it is being developed. This model, of course, varies highly between programmers. Nevertheless, some tailorable high-level medium would be helpful to the programmers in refining his understanding of the code.

The programmer subdivides the program into modules, or parts with common meaning or intent, and analyses them in terms of their function and contribution to the program. The programmer follows this modularizing method until the lowest details of the code are understood. At this point he is ready to formulate a solution.

It would be useful to help the programmer in identifying semantic modules. This could be achieved by supporting possibly related user queries, such as letting the user find a piece of code where a certain variable is modified, where it is being declared, where a certain data structure is changed, or updated.

The programmer also needs a method of storing and retrieving subgoals which could either be provided for him, or developed by him as his understanding increases.

Most programmers still prefer to use some comments. For this reason it is useful to allow the programmer to choose one of two options: view with comments, view without comments. This could be

implemented via a link between the comment and the part of code to which it belongs. Likewise, it is straightforward to provide for user control of indentation and a mechanism for systematic replacement of variable names in cases where a programmer can devise a new name with more mnemonic relevance than an existing one.

**Appendix A**

A maintenance environment should allow for :

{1} Modification, storage and retreval of a general definition of what the program does.

{2} Modification, storage and retreval of subgoals of the modification task.

{3} Modification, storage, and retrieval of semantic model.

{4} Identification of semantic modules.

{5} View code with or without comments.

{6} Control of indentation.

{7} Systematic name substitution.

**References**


1. J. R. Barclay, ‘‘The Role of Comprehension in Remembering Sentences,’’ *Cognitive Psychology*, vol. 4, pp. 229-254, 1971.
2. J. D. Bransford and J. J. Franks, ‘‘The Abstraction of Linguistic Ideas,’’ *Cognitive Psychology*, vol. 2, pp. 331-350, 1971.
3. O. J. Dahl, E.W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.
4. J. G. Greeno, ‘‘The Structure of Memory and the Process of Problem Solving,’’ 37, University of Michigan: Human Performance Center, 1972.
5. Tom Love, *Relating Individual Differences in Computer Programming Performance to Human Information Processing Abilities*, University of Washington, 1977. Ph.D. Thesis
6. P. R. Newsted, *FORTRAN Program Comprehension as a Function of Documentation*, School of Business Administration, University of Wisconsin, Milwaukee, Wisconsin, 1973.
7. G. H. Okimoto, ‘‘The Effectivenss of Comments: A Pilot Study,’’ IBM SDD 01.1347, July 27, 1970.
8. D. Parnas, ‘‘On the Criteria to be Used in Decomposing Systems into Modules,’’ *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.
9. Ben Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Little Brown and Co., Boston, Massachusetts, 1980.
10. B. Shneiderman and D. McKay, ‘‘Experimental Investigations of Computer Program Debugging and Modification,’’ *Proceedings 6th International Congress of the International Ergonomics Association*, College Park, Maryland, July 1976.
11. B. Shneiderman, ‘‘Exploratory Experiments in Programmer Behavior,’’ *International Journal of Computer and Information Sciences*, vol. 5, no. 2, pp. 123-143, June 1976.
12. B. Shneiderman, ‘‘Measuring Computer Program Quality and Comprehension,’’ *International Journal of Man-Machine Studies*, vol. 9, 1977.
13. L. Weissman, ‘‘Psychological Complexity of Computer Programs: An Experimental Methodology,’’ *ACM SIGPlan Notices*, vol. 9, 1974.
14. W. Wickelgren, *How To Solve Problems*, W. H. Freeman, San Francisco, 1974.