DRAFT - 2

Program Comprehension

Spencer Rugaber
Georgia Institute of Technology
May 4, 1995

I. Introduction

A. Definition of Program Comprehension

Program comprehension is the process of acquiring knowledge about a computer program. Increased knowledge enables such activities as bug correction, enhancement, reuse, and documentation. While efforts are underway to automate the understanding process, such significant amounts of knowledge and analytical power are required that today program comprehension is largely a manual task.

B. Motivation

Program comprehension is an emerging interest area within the software engineering field. Software engineering itself is concerned with improving the productivity of the software development process and the quality of the systems it produces. However, as currently practiced, the majority of the software development effort is spent on maintaining existing systems rather than developing new ones. Estimates of the proportion of resources and time devoted to maintenance range from 50% to 75% [1][2].

The greatest part of the software maintenance process, in turn, is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities. These involve reading the documentation, scanning the source code, and understanding the changes to be made [3].

The implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of comprehending existing programs.

C. Terminology and Relationship to Other Activities

Other terms are sometimes used to describe activities related to program comprehension. Chikofsky and Cross have given standard definitions for them [4]. For example, "*reverse engineering* is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."

A variation of reverse engineering is *design recovery.* In design recovery, advantage is taken not only of the source code, but also of other information such as domain knowledge, documentation, in-line commentary, and mnemonic variable names. Its goals are similar to those of reverse engineering, while concentrating particularly on the discovery of design decisions and their rationale.

A closely related term is *reengineering.* Whereas reverse engineering moves from program code to a higher level of abstraction, reengineering uses the increased understanding to reimplement the code in a new form. The terms *renovation* and *reclamation* are also used to describe this activity. The need for reengineering may arise due to 1) changes in the operational environment, such as moving from a centralized computational setting to one where use is decentralized, 2) degradation to system structure due to long-term maintenance, or 3) so many alterations and enhancements to a system that the existing architecture is no longer appropriate.

The term reengineering is also popular in another context: *business process reengineering*. In this case, the target of the reengineering is an organization instead of a software system, but the goal is still reformulation. Smith and McKeen [5] give the following definition: “At a basic level, re-engineering means radically redesigning the way an organization performs its business to achieve dramatic improvements in performance.” Of course, to the extent that a company’s way of doing business is expressed by its software systems, software reengineering and business process reengineering are intimately linked.

Yet another related term is *restructuring.* It is sometimes desirable to reformulate a program without first abstracting it to a higher level. For example, older software, typically written in versions of Cobol or Fortran that did not contain modern control structures, makes heavy use of **GOTO** statements. Readability can sometimes be improved by replacing uses of **GOTO** with their modern equivalents, such as structured conditional and looping constructs [6]. This process of reformulating the program without raising the level of abstraction is called restructuring, and commercial tools are available for many languages to automate the process.

II. Why is Program Comprehension Difficult?

Program comprehension is difficult. It is difficult because it must bridge different conceptual areas. Of particular importance are bridges over the following five gaps.

- The gap between a problem from some application domain and a solution to it in some programming language.
- The gap between the concrete world of physical machines and computer programs and the abstract world of high level design descriptions.
- The gap between the desired coherent and highly structured description of a system as originally envisioned by its designers and the actual system whose structure may have disintegrated over time.
- The gap between the hierarchical world of programs and the associational nature of human cognition.
- The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application.

A. Application Domain and Program

Programs are solutions to problem situations from some application domain. There may or may not be hints in the program about the particular problem. Hints can take the form of mnemonic variable names and in-line comments. The hints are inherently informal and tend to be out-of-date with respect to the program. Because of this, totally automatic program comprehension tools are restricted to working with the formal program text. It is the job of the person trying to understand the program (also called the *reverse engineer* or *program reader*) to reconstruct the mappings from the application domain to the program. This of course requires knowledge, not only of programming, but also of the application domain. It is no surprise, therefore, to find that most automatic tools are restricted to analyzing the program text and do not address the application domain.

B. Physical Machines and Abstract Descriptions

Computer programs are incredibly detailed. In essence they control the values of million of bits of memory inside of a computer. One of the jobs of the reverse engineer is to decide, from all this detail, which are the important concepts. This process is called *abstraction*; the reverse engineer must create an abstract representation of the program from the mass of concrete details. The abstraction process is not linear. That is, a given section of a program may be a part of several abstractions. The abstractions are said to be *interleaved* [7], and, because of this, the designer’s plan is *delocalized* in the section [8]. Typically there is no documentation in the source code of the interleaving.

C. Coherent Models and Incoherent Artifacts

When a program is originally constructed, there is a coherent structuring of details. The process that creates the structuring is called *design*. A large variety of design methods and representation techniques have been developed to aid this process [9]. Although programming languages have some features intended to facilitate abstraction and structuring, the higher-level design representations may have been lost or allowed to become out-of-date by the time program comprehension is required. More importantly, through maintenance activities such as porting, bug fixing, and enhancement, the original structure of the program may have deteriorated [10]. That is, it is the job of the person trying to understand the program to detect the purpose and high level structure of a program when the original purpose of the program may have changed and where, in fact, the program may now serve to accomplish several purposes.

D. Hierarchical Programs and Associative Cognition

Computer programs are highly formal. They obey strict rules that limit the expression of ideas and that control how the ideas effect the computer when they run. The two types of rules, syntax and semantics, are organized hierarchically with broad concepts like *program* and *function* defined in terms of narrower ones like *declaration* and *expression*. In the formal world, the meaning of a syntactically correct program determines the output that is produced when a specific input is presented. Human cognition, to the extent that it is understood at all, seems to work associatively. Raw data are perceived, patterns are detected, and abstractions (also called *chunks* [11]) are constructed relating them. The process of human understanding is controlled by expectations derived from the application domain of the program and the large body of programming knowledge held by the reverse engineer: knowledge of the programming language, typical programming practices, algorithms, and data structures. A program is understood to the extent that the reverse engineer can build up correct high level chunks from the low level details evident in the program.

E. Bottom-up Program Analysis and Top-down Model Synthesis

When an experienced reverse engineer looks at a program, he or she detects patterns that indicate the intent of some section of code. Low level patterns are part of higher level constructs intended to accomplish larger purposes. In this case, the process of analyzing a program proceeds bottom-up [12]. At the same time, the programmer has some idea of the overall purpose of the program and how it might be accomplished. As the program is perused, the overall concept is refined into a more complete description by adding lower level details [13]. This synthesis process proceeds top-down. The difficulty is that both of these activities need to proceed at the same time, in a synchronized fashion [14].

III. Program Comprehension by People

Most program comprehension is currently done by humans. In order to understand the process, it is important to look at the human factors involved in comprehension. This study is called *software psychology.* As a result of these efforts, a variety of models of the human program comprehension process have been proposed. Ultimately, tools will be developed to support them. For further information on the matter, the reader is referred to surveys by von Meyrhauser and Vans [15], Robson et al. [16], and Paul et al. [17], and to Quilici’s recent empirical study [18].

A. Software Psychology

The study of software psychology was pioneered by Shneiderman. It attempts to discover and describe human limitations in interacting with computers. Shneiderman [19] defines software psychology as the “study of human performance in using computer and information systems.” It uses the techniques of experimental psychology to analyze aspects of human performance in computer tasks. It also applies the concepts of cognitive psychology to the cognitive and perceptual processes involved in computer interaction. 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.

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. Of most relevance to software maintenance are experiments on program comprehension. Program comprehension is studied via experiments requiring the memorization and reconstruction of programs. A memorization/reconstruction task consists of studying a program and then reconstructing it from memory. It has been found that experience of the subjects plays a vital role in this task. An experiment by Shneiderman [20] 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 explanation is that as subjects gain experience in programming they improve their capacity to recognize meaningful program structures thus enabling them to recode the syntax into a higher-level internal semantic structure.

After the subject completes the reconstruction task, the experimenter analyzed the reconstructed code in terms of information chunks, which give insight to the kinds of internal information structures contained in the code. It is in terms of these internal structures that the code is interpreted, or understood, by the subjects. Shneiderman identifies these structures in his model.

1. Shneiderman's Model of Program Comprehension

The majority of the program maintenance tasks are founded on 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 nature of the algorithms and data, and high-level comprehension of 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 algorithms and data. 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.

Experienced programmers possess a network of multi-leveled concepts in their long-term memories. Some of these concepts, which are extracted from experience and are independent of programming language or environment, comprise the programmer's semantic knowledge. Semantic knowledge consists of concepts such as what an assignment statement does, how a stack is implemented and 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 details of different programming languages and systems such as the proper positioning of semicolons, the legal syntax for assignment and conditional statements, available data types, and other features of the language or environment.

Shneiderman 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, programmers form an idea of the program's purpose. They then recognize lower-level structures such as algorithms for sorting and searching, familiar streams of statements, or others. Finally, they reach an understanding of what the program does as well as how it does it. This understanding is represented in some internal form. The 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 contexts.

2. Examples of Software Psychology Experiments

Commenting. The influence of comments on program understanding is not resolved. Studies of short programs [21] show that comments in code interfere with the process of understanding, require more filtering when reading the code, and, if not up-to-date, can 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 cited by Shneiderman [22] [23] show that functionally descriptive comments do facilitate faster conversion of code to internal semantic structure, while non-descriptive comments hinder it. Functionally descriptive comments are high-level comments that describe actions or effects not obvious when viewing the code. Low-level non-descriptive comments that restate the function of the code hinder program understanding by unnecessarily interrupting the subject's thought process.

Variable Names. The use of mnemonic variable names contributes to program comprehension [24]. The mnemonics, however, have to be such that they add semantic information relevant to the code. It is most likely that mnemonics have different meanings to different programmers. Allowing for systematic substitution of variable names according to individual programmer's specifications could improve comprehension. Having meaningful mnemonics reduces the programmer's short term memory load, making comprehension easier.

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 [23]. Love [25] shows that indentation does not improve understanding for short FORTRAN programs, and Shneiderman and McKay [26] show that indented long programs are more difficult to read because deep indentation can cause lines to split in order to accommodate margins.

B. Models of Comprehension

Based upon software psychology studies, a variety of models have been proposed for the process of program comprehension. Two are described here to illustrate the issues involved in modeling this complex behavior. They were developed by Ruven Brooks and Elliot Soloway. At the highest level, the basic structure for both cognitive models consists of four components:

- The target system to be comprehended. This consists of all the information sources available to the understander, such as source code and supporting documentation.
- The knowledge base that encodes the understander's experience and background knowledge used in the comprehension task. This knowledge base is either internal (in the understander's mind) or external, for example in the reference manual for a programming language.
- The mental model that encodes the current state of understanding of the target program. It is constantly being updated in the course of comprehension.
- An assimilation process that interacts with the other three components to update the current state of understanding.

The differences in the two models of human comprehension are in the terminology describing the contents of the knowledge base and in the approach that each adopts for the assimilation process.

While the models contain mechanisms that utilize both top-down and bottom-up approaches to the comprehension process, Brooks' model emphasizes the top-down approach, while Soloway's model is more bottom-up.

1. Brooks' Model

Ruven Brooks' model deals with the comprehension of completed programs [27][28]. It has its basis in areas outside of computer science, such as thermodynamics problem solving, physics problem solving, and chess. The model was initially created to explain four major sources of variation observed in the act of program comprehension.

- The functionality of the program being understood. How do programs that perform different computations vary in comprehensibility?
- The differences in the program text. Why do programs that are written in different languages differ in comprehensibility, even though the same calculation is performed in each?
- The motivation the understander has to comprehend the program. Why does the comprehension process vary depending on whether the motivation is to debug the program or enhance it?
- Individual differences between understanders' abilities to comprehend a program's purpose. Why does one understander find a program easier to comprehend than does another?

To account for these four areas of variation, Brooks created a model based on three main ideas.

- The programming process is the construction of mappings from a task domain, through one or more intermediate domains, to the programming domain.
- The comprehension process of that program is the reconstruction of all or part of those mappings.
- The reconstruction process is expectation-driven by the creation, confirmation and refinement of hypotheses. These hypotheses describe the various domains, and the relationships between them.

The comprehension process is one of (re-)creating the set of mappings that were used to develop the program. The mappings are first expressed as hypotheses. An example of a high level hypothesis is: "This program produces invoices." This hypothesis maps the task domain, invoicing, to the programming domain, the program itself.

At the very start of the understanding process, the understander forms a primary hypothesis, which is a global, abstract description of what the understander thinks the program does. It is formed as soon as the understander obtains any information at all about the purpose of the program. For example, hearing the program name usually provides enough information to form the primary hypothesis. The primary hypothesis then produces a cascade of subsidiary hypotheses. This cascading is done depth-first, with the decision of which hypothesis to pursue based on the understander's motivation for comprehending the program.

The cascading continues until it produces a hypothesis that is specific enough that the understander can verify it against the program code and/or supporting documentation. Specifically, the understander begins the verification of a hypothesis when the hypothesis deals with operations that can be associated with visible details found in the program code. The term *beacons* describes those details that show the presence of a particular structure or operation. As an example, a typical beacon for the hypothesis ‘sort’ is a pair of loops, inside of which there is a section of code where the values of two elements are compared and conditionally interchanged.

Beacons are important because they are the first link between the top-down hypotheses and the actual program text. Their existence has been verified by experiments done by Wiedenbeck [29]. The method used in these experiments was memorization and recall, under the assumption that a high recall of some part of a program, after brief study, indicates that the recalled part is important in understanding the program. The tested hypothesis was that experts could locate these parts more efficiently than novices. This effect was confirmed, as well as the effect of experts recalling beacons lines more successfully than non-beacon lines (77% beacon versus 47% non-beacon). This result was opposite for the novices, they recalled non-beacon lines better (13% beacon versus 30% non-beacon). This effect is explained by two factors. First, syntactic markers, such as “begin” and “end” were placed one per line, and were considered non-beacons. Second, since the novices recalled lines at the beginning of the program more effectively than later lines and many of the beacon lines were fairly deeply embedded within the program, the net result was an increase, percentage-wise, of non-beacon lines to beacon lines.

As stated earlier, Brooks’ theory attempts to explain four sources of variation in the act of comprehension.

- The functionality of the program. The intrinsic complexity of the task domain (e.g. nuclear physics versus invoicing) causes the higher level hypotheses to be more complex and can result in the use of a greater number of intermediate domains between the task domain and the programming domain. Also, documentation explaining these intermediate domains is rarer than documentation covering the original program task.
- The program text. The code and supporting documentation affect the ease that beacons are located and the ease of binding the actual source code to hypotheses. Also, features of the language effect the confirmation of beacons. For example, the hypothesis that “The variable ‘PI’ contains 3.1415” is easy to confirm in the C language if PI is defined using a **const** statement. It is not so easy in a language without constant declarations, because even when 3.1415 is put into ‘PI’ at the beginning of the program, one must confirm that ‘PI’ is never changed.
- The understander’s task. The motivation behind the understander affects the strategy used in creating and following subsidiary hypotheses. For example, debugging an output format error causes entire subsidiary hypotheses, such as those dealing with input or computation, to remain unexplored because of the lack of relevance to the task at hand.
- The understander’s individual abilities. The knowledge the understander has about domains affects the process on all levels. Specifically, task domain knowledge affects the quality of the primary and higher-level hypotheses. Programming domain knowledge affects the lower level bindings and beacon location process.

2. Soloway’s Model

Elliot Soloway’s model also deals with the comprehension of completed programs, but divides the knowledge base and the assimilation process differently [30][31]. In Soloway’s terminology, to understand a program is to recover the intention behind the code. *Goals* denote intentions, and *plans* denote techniques for realizing intentions. Plans work as rewrite rules that convert goals into subgoals and finally into program code. Program comprehension is defined as the process of recognizing plans in code, combining these plans by reversing the rewrite rules to form subgoals, and combining the subgoals into higher level goals.

The knowledge base used in this model contains many parts including the following.

- Programming language semantics. This deals with the understander's knowledge of the language in which the target program is written.
- Goal knowledge. This is the encoding of the understander's set of meanings for computational goals. The goals are encoded independently of the algorithms and the languages that implement them.
- Plan knowledge. This is the encoding of solutions to problems that the understander has solved or understood in the past. These solutions are low level components and include those universally known to programmers. Plan knowledge also includes plans the understander has acquired of domain-specific knowledge.
- Efficiency knowledge. This is how understanders detect inefficiencies and the influence that efficiency issues have on programming code and plans.
- Problem domain knowledge. This is the understander's knowledge of the world from the application domain to the computational domain and those domains in between.
- Discourse rules. This is the knowledge of certain programming conventions, which allow the understander to attach greater meaning to aspects of the source code and/or documentation than is ordinarily possible. An example of a discourse rule is "If a variable name forms a word, the meaning of that word is somehow related to the purpose of that variable".

The supporting experiments that have been performed concern behavior and events that last for short periods of time[32]. For example, events such as reading a line of code, formulating a question, or stating a hypothesis are all events that have been looked at in detail. It is clear that one of the advantages of this approach is that experimental evidence to support the theory is relatively easy to obtain. The empirical studies have a shown the existence of discourse rules and plans, and even the concept of a de-localized plan, a single plan that has been physically implemented in separate sections of source code to explain varying levels of complexity in comprehension [33][34].

The major implications of this model concern the building, use, and manipulation of the knowledge base. The element of domain knowledge is similar to Brooks' domains. The elements of goal and subgoals similarly correspond to the task and intermediate domains discussed earlier. The formally defined plans correspond to the mappings between the low level domains that Brooks mentions. But the mappings between high level domains are not addressed by plans. This is a direct consequence of the formal, rigid structure of plans: they have gained expressive power at the low level by sacrificing the power needed to express high level domain relationships.

IV. Foundations of Automated Program Comprehension

Computers are much more rigorous and formal than humans. To understand how a computer program could comprehend other programs, the underlying formal basics of program analysis must be appreciated. This section presents a compendium of ideas related to program analysis. It serves both as a vehicle to present the ideas and as an indication of the variety of approaches undertaken so far. It proceeds from straightforward textual analysis through increasingly more complex static approaches to the dynamic analysis of executing programs.

A. Textual Analysis

Programs are, at their most basic, sequences of characters, and one crude measure of the comprehensibility of a program is just the number of characters that it contains. A more useful measure, however, is to group characters into lines and use the number of lines of code as an indication of the size of a program. In fact, the single factor that best predicts the amount of effort required to comprehend a program is the number of lines it contains. Other factors, such as the control flow complexity and the pattern of variable usage are secondary. Programs range in size from *one-liners* in APL to systems contains millions of lines of code, such as ones to control telephone switches.

B. Lexical Analysis

Just as with natural languages, sequences of characters in program text fall into certain lexical categories. Instead of nouns and verbs and adjectives, however, the lexical units comprising programs consist of identifiers, operators, keywords, strings, numbers, and punctuation marks. Lexical analysis is the process of decomposing the sequence of characters in a program's input file into its constituent lexical units. Once lexical analysis has been performed, various useful representations of program information are enabled. Perhaps the most common is the cross reference listing. In this report, program identifiers are listed along with the numbers of the lines on which they occur. A program maintainer can then easily locate and examine code segments affected by some modification to the program text. Many compilers produce cross reference listings as a regular part of the compilation process.

One of the most popular software complexity metrics is also enabled by lexical analysis. This metric was devised by Halstead [35] and uses the total number of identifiers, total number of operators, number of unique identifiers and number of unique operators in order to compute various measures including difficulty in comprehension and effort to program.

Lexical analysis is, naturally enough, performed by the lexical analyzer (or lexer) part of a programming language's compiler. Typically, it uses rules describing lexical program structure that are expressed in a mathematical notation called *regular expressions*. In fact, it is commonplace today to build lexical analyzers automatically using tools called *lexical analyzer generators* such as **lex** or **flex** [36]. These tools take as input a set of regular expressions and produce either a set of tables modeling a finite state acceptor for the language to be interpreted or actual code that does the analysis directly.

The result of lexical analysis is a stream of tokens (or *lexemes*) and a set of tables, such as a compiler's symbol table, that describe properties of the lexemes. The lexical analyzer produces enough information to easily answer questions like "what is the average length of variable names?"

C. Syntactic Analysis

The next most complex form of automated program analysis is syntactic in nature. Just as we parse natural language text into phrases and sentences, so do compilers and other tools determine the expressions, statements, and modules of a program.

Syntactic analysis is performed by a parser. Here, too, the requisite language properties are expressed in a mathematical formalism, in this case called a *context free grammar*. Usually, these grammars are described in a stylized notation called *Backus Naur Form* (BNF) [37] in which the various program parts are defined by rules in terms of their constituents. It is also the case with syntactic analysis that parsers can be automatically constructed from a description of the grammatical properties of a programming language.

Two types of representation are used to hold the results of syntactic analysis. The more primitive of the two is called a *parse tree*. It is similar to the parsing diagrams used to show how a natural language sentence is broken up into its constituents. However, parse trees contain details unrelated to actual program meaning, such as the punctuation, whose sole purpose is to direct the parsing process. Removal of these extraneous details leads to a structure called an *abstract syntax tree* (AST) that is the basis of most sophisticated program analysis tools. The AST contains just those details that relate to the actual meaning of a program.

Because an AST is a tree, it can be traversed or *walked*. That is, nodes of the tree can be visited in a pre-set sequence, such as depth-first order, and the information contained in the node delivered to the analyzer. This approach serves as the basis of many tools in which the analyst requests desired knowledge as a high level query expressed in terms of the node types. A standard tree walker then interprets the query and delivers the requested information.

D. Control Flow Analysis

Once a program's syntactic structure is determined, it is possible to perform control flow analysis (CFA) on it [38]. There are two forms. *Intraprocedural* analysis provides a determination of the order in which statements can be executed within a subprogram. *Interprocedural* analysis determines the calling relationship among the program units.

Intraprocedural analysis proceeds by constructing a *control flow graph* (CFG) that is similar to a flow chart. To construct a CFG, first the basic blocks of the subprogram must be determined. A *basic block* is a maximal collection of consecutive statements such that control can flow in only at the top and leave only at the bottom via either a conditional or an unconditional branch. That is, if the first statement in the block executes, then all of the statements execute. A basic block corresponds to a node in the control flow graph. Arcs indicate possible flows of control. Arcs can be forward, usually indicating a branch, or backward, indicating a loop.

A CFG can be directly constructed from an AST by walking the tree to determine basic blocks and then connecting the blocks with control flow arcs. The control flow graph gives an abstract picture of the ways in which a subprogram could execute without being cluttered by the details of the statements in each of the basic blocks.

Another popular metric can be computed from the control flow graph. This metric is called *cyclomatic complexity* and was devised by McCabe [39]. It is a measure of the amount of branching in a program. The assumption is that in two programs that are otherwise similar, the one with the greater amount of branching is the more complex.

Interprocedural control flow analysis is a determination of which routines can invoke which others. This information is often displayed in a *call graph*, where the main routine is at the top, and a routine is connected by arcs from above to all the routines that can call it and by downward arcs to all routines that it calls. In the absence of procedure parameters and pointers, this analysis is trivially computed from the AST by looking for statements that can call others. Procedure parameters and pointers allow the possibility that the actual routine called will not be known until run-time. In these situations, interprocedural analysis is often performed conservatively; that is, all routines that by static analysis can possibly be the value of the procedure parameter or pointed to by the procedure pointer are listed as possible targets of the call. This, of course, may lead to the situation where so many routines are listed that the analysis is effectively worthless.

E. Data Flow Analysis

Although control flow analysis is useful, there are many questions that it cannot answer, such as, which statements may be affected by the execution of a given assignment statement? To answer this kind of question, an understanding of definitions (*defs*) and references (*uses*) is required. The usual way that a variable is defined is if it occurs on the left hand side of an assignment statement. Note that a given variable may be defined by numerous such statements. A use of the variable is when its value is referenced by some other statement, for example, when it appears as a parameter to the call of a function or as an operand in an arithmetic expression.

Data flow analysis (DFA) is concerned with answering questions related to how definitions flow to uses in a program [38]. Data flow analysis is significantly more complex than control flow analysis. In particular, whereas CFA merely has to detect the possibility of loops, DFA has to describe what might happen to the variables inside the loop body. However, significant additional power derives from the additional effort to perform DFA. For example, code that can never execute, variables that might not be defined before they are used, or statements that might have to be altered when a bug is fixed are all examples of tasks enabled by data flow analysis.

In interprocedural data flow analysis the graph of def/use dependencies is extended across procedural boundaries. Problems arise when a procedure is called with arguments two or more of which correspond to the same memory location. This is called an *alias*, and aliases can also arise when pointers are used. One simple representation of interprocedural DFA information is the structure chart [40]. A *structure chart* is a call graph in which arcs are annotated with the names of the formal parameters and an indication of whether the arc is supplying values to the called procedure or returning them.

F. Program Dependence Graphs

A further refinement of DFA is the construction of a *program dependence graph* (PDG). PDGs were developed by researchers interested in converting programs to run on machines with parallel architectures. In a PDG control and data flow dependencies are treated together in the same representation. This uniform treatment of data and control is convenient in situations which would otherwise require both a DFG and a CFG. In addition to allowing a more uniform treatment, PDGs also have structural properties that make them useful in program comprehension. For example, rather than representing control flow through basic blocks as in a CFG, a PDG represents regions of control dependence. For many applications, regions are conceptually more appropriate than basic blocks because they are canonical. No two regions depend upon the same control conditions; whereas often many basic blocks in a CFG depend upon the same control conditions. Further, regions may be factored since they are defined in terms of dependence. That is, if two regions R1 and R2 have common control conditions, there must exist another region R3 that depends upon exactly these common control conditions. R1 and R2 are then said to be *control dependent* upon R3.

A popular extension to the PDG is to represent data dependence using *static single assignment* (or SSA) form. SSA form establishes that each use of a variable is reached by exactly one definition of that variable, and that no variable is defined more than once. Determining answers to def/use questions can often be done significantly more efficiently using SSA form, as each use is reachable by a unique definition [41].

G. Slicing

Another popular derivative of data flow analysis is provided by slicing. Slicing was introduced by Weiser and has served as the basis of numerous program comprehension tools [42]. The *slice* of a program for a particular variable at a particular line of the program is just that part of the program responsible for giving a value to the variable at that spot. Obviously, if while debugging you determine that the value of a variable at a particular line is incorrect, it is easier to search for the faulty code by looking at the appropriate slice than by examining the entire program.

H. Cliche Recognition

One further elaboration of static program analysis has recently been proposed. It involves searching the program text for instances of common programming patterns. These patterns are called *cliches* (or idioms), and several research tools provide cliche libraries against which they automatically perform searches [43]. An example of a cliche is a pattern describing loops for performing a linear search. Obviously, the value being searched for, the data structure being searched, and possibly the mechanism for determining the match are all parameters of the cliche, thus complicating the detection process. Moreover, there are a variety of ways for programming a linear search even if variations due to parameters are ignored. Thus, cliche recognition is a difficult research problem, but the abstraction power it provides promises to make it a useful program comprehension technique.

I. Abstract Interpretation

One final approach to static program analysis should be mentioned here. Just as the syntactic properties of a program can be specified with a grammar expressed in BNF, so too can the semantic properties be provided with a mathematical technique called *denotational semantics* [44]. In this approach, the meaning of a program is expressed in terms of various data types called *semantic domains*. For example, program state, the bindings between variables and values, is denoted by a table-like domain. The meaning of a syntactic program construct like a *statement* is then given by a function that describes what happens to the corresponding domains. For example, the meaning of an assignment statement is a function that maps from the state before the assignment was executed to the state afterwards.

It is possible to use the denotational semantics to perform static program analysis. In this case, alternative functions are defined on variations of the original semantic domains. So, for example, if we wished to know only whether a given variable was changed by a subprogram, we might interpret the semantic function for the assignment statement to refer only to that variable and instead of worrying about its possible values, we can use only a single Boolean variable to indicate whether or not it has changed. This process of re-interpretation is called *abstract interpretation* [45] and has recently become popular for a variety of analysis tasks particularly in the area of logic programming languages such as Prolog.

J. Dynamic Analysis

The analysis techniques described so far have all been static; that is, they are performed on the source code of a program [46]. It is also possible to gain increased understanding by systematically executing a program. This process is called *dynamic analysis* and is most frequently used when we are trying to understand the performance and correctness properties of a program where they are called, respectively, *profiling* and *testing*. A statement-level profiler determines the number of times each statement is executed; a procedure-level profiler does the same thing for procedure calls and returns. Some profilers work by instrumenting a program, inserting extra code to do the counting, and others construct an approximation by periodically interrupting the executing program to determine what it is currently doing and then constructing a statistical model.

Of course, testing is the most common form of dynamic analysis, and a variety of techniques exists for making sure that a test suite thoroughly exercises (*covers*) a program [47]. *Statement coverage* assures that every statement is executed. *Branch*, *condition*, and *path coverage* measure the extent to which all branches, conditions, and paths are executed. Numerous variations exist, many of which make use of the other techniques described in this section.

K. Partial Evaluation

One final form of dynamic analysis deserves mention. Let us say that we are trying to understand a complicated real-time system that implements a complex state machine architecture by referring to a collection of global variables with a series of nested conditional statements. Systems such as this are common in the telecommunications industry. Suppose further that we are trying to understand some particular anomalous program behavior that arises only under certain circumstances;

for example, only when some of the global variables have specific values. The traditional way to understand this situation is to “play computer”; that is, to mentally simulate execution, determining the flow of control by evaluating the predicates of the conditional statements in terms of the known values of the global variables.

Of course, this process breaks down when the program gets large. An approach called *partial evaluation* has been devised to address problems like this [48]. A partial evaluator is a software tool that takes as input a program and values for certain of the program’s input parameters. It produces as output a smaller program equivalent to the original on those parameters. That is, the partial evaluator executes as much of the program as it can, where possible replacing program statements with the values computed by them. It is like a paint-by-numbers toy, where the manufacturer has partially evaluated the painting by supplying an outline while still leaving it to the consumer to produce the final picture.

In the case of a telecommunications system, input parameters are really indications of the state of the switch hardware, what lines are in use, etc. These correspond in a natural way to the global variables defining the system state. Partial evaluation is currently a research topic, but it promises to provide help in understanding such programs.

V. Program Comprehension Tools

There are many tools that have been developed to aid program comprehension. Some of them are available commercially, while others are still research prototypes.

A. Commercial Tools

1. Compilers

Compilers, of course, are the most commonly used program understanding tools. There are several senses in which a compiler can be thought to understand a program. First, the compiler must understand the program well enough to translate it into the language of the underlying machine. Then there are optimizations. Optimizing compilers must make sure that the transformations they apply to a program carefully preserve all of the program’s semantics. Compilers also must understand errors. Some compilers even attempt to guess what the programmer really meant by an erroneous construct and to supply a corresponding correction. And, of course, compilers also supply all sorts of auxiliary information such as cross reference tables, warnings about portability problems, and possible anomalies such as uninitialized variables. And, finally, compilers provide various additional services such as augmenting the executable version of a program with information to support profiling and debugging.

2. Restructurers and Beautifiers

There is a class of commercial tools specifically designed to improve the comprehensibility of programs. This class includes *beautifiers* and *restructurers* [49]. Their input is typically an older program, either one written in an early version of a language that did not include modern control structures or one that has undergone sufficient maintenance that its original structure and purpose are no longer obvious.

Restructurers modernize a program’s control flow patterns, replacing “spaghetti” code featuring numerous **GOTO** statements with conditional statements and loops. While this level of analysis is fairly superficial, it can help improve comprehensibility by localizing related code segments and suggesting higher level abstractions.

Beautifiers are similar to restructurers but have knowledge of program layout issues such as indentation, bracketing conventions for compound statements, and use of whitespace in expressions. Beautifiers are particularly useful for standardizing the appearance of a large program that has undergone maintenance by different programmers using various styles.

3. Translators, Vectorizers, and Parallelizers

Other kinds of commercial tools exist to convert a program to a different form. While not specifically aimed at improving comprehensibility, by allowing an algorithm to be expressed in an alternative form, the tools may enable simplifications that do have that effect.

The first example is the language-to-language *translator* (for example, **f2c**, [50]). If the two languages are at the same level of abstraction, then it is likely that the transformed code is less rather than more comprehensible than the original. This happens because of the superficial level of understanding that the typical commercial translator has of the source program. If the target language is at a higher level or particularly well adapted to an application area, then a successful translation can lead to a significantly smaller or more modular program [51]. For example, an SQL version of a Cobol program can take advantage of features built into database utilities to remove voluminous Cobol file manipulation code.

Of a similar nature are *vectorizers* and *parallelizers* [52]. These tools are capable of taking advantage of regularities in an algorithm to produce code for vector hardware boxes and parallel machines, thereby producing answers more efficiently. For example, many loops can be replaced by a single statement acting simultaneously on several elements of an array. The resulting program is consequently reduced in size from the original and presumably easier to understand.

4. CASE Tools

Computer Aided Software Engineering (CASE) is a segment of the computer software market originally promoted as helping with the initial construction of large software systems by teams of developers [53]. CASE tools typically provide a variety of graphical editors for expressing high level designs, consistency checkers for detecting problems, and, in some cases, code generators for actually producing programs. Recently, however, CASE vendors have realized that the same types of diagrams previously manually drawn by developers can prove useful in understanding existing code. And, in many cases, the diagrams can be automatically constructed from the code.

Probably the most common diagram is the structure chart [40]. It indicates which subprograms can invoke which others and the names and types of the arguments passed among them. Other types of available diagrams include the dataflow diagram[1] indicating the major software modules and data repositories and how data and control information flow among them [54], and the *entity-relationship diagram* [55] used to describe the major external sources and sinks for data and the modules that use them. Expect CASE tools to become more sophisticated in the future as more types of diagrams and further analyses are added.

5. Program Analysis and Transformation Tools

There is a class of tools that has recently entered the marketplace that differs significantly from the ones mentioned above. Tools in this class are based on the idea of a *wide-spectrum language* for specifying program analyses and transformations [56]. A wide-spectrum language contains features of other classes of languages, typically imperative, functional, object-oriented, and logic languages. The added power provide by the language enables terse expression of queries about program features. One example of this class of language is **Refine** [57], which has been successfully used on a variety of ambitious program understanding projects.

Refine is the basis for a collection of commercial tools called the Software Refinery [58]. The Software Refinery consists of six components that together provide a comprehensive toolkit. One piece consists of a set of language-specific browsers and analyzers that have a deep understanding of a particular language. The browsers/analyzers are also capable of producing a variety of reports including those mentioned above, such as cross reference listing and structure chart. For example, the **Refine/C** tool provides source code, structure chart, and dataflow diagram browsers, as well as a variety of reports describing the use of names in C programs.

The second and third pieces of the Software Refinery are used for building language-specific tools like **Refine/C**. The second is a parser generator, and the third is a user interface builder. These tools express their results in **Refine**, and the **Refine** compiler is the fourth tool. **Refine** includes pattern matching features that support the querying of the abstract syntax trees built by the language-specific parsers. The AST, the symbol table, and the results of any other analyses are stored in an object-oriented repository to which Refine language statements can easily access. This repository is the fifth component of the Software Refinery's architecture. The sixth is a set of built-in language-independent analyzers for constructing common representations, such as structure charts.

The Software Refinery has been described in some detail both because it is one of the most advanced tools commercially available and because its complexity is indicative of the amount of effort required to perform even the modest analyses it does.

B. Research Tools

This section gives a quick look at a variety of tools currently under development by the research community. Its purpose is to show the breadth of approaches undertaken without making any claims to completeness.

[1] Not to be confused with dataflow analysis described earlier.

1. Gen++

Gen++ is a C++ *analyzer generator* [59]. That is, its input is a description of a specific program query or analysis that is desired of one or more C++ programs. Its output is an analyzer that when given a C++ program, performs the given query on the program. The language in which queries are given contains a variety of high-level constructs both for specifying tree walks and for formatting output. The tool's strength comes from its in-depth knowledge of C++, a notoriously difficult language to build tools for, and from its comprehensive AST representation that contains the data extracted during analysis.

2. CIA and CIA++

CIA stands for the C Information Abstractor [60]; **CIA++** is the more recent C++ variant [61]. Both tools work by placing the results of a standard analysis into a relational database. Then queries on the database can give answers to many of questions concerning a program. Both tools begin with a program data model containing entities and relationships describing the structure of typical programs. Examples of typical entities are modules, global variables, and files; typical relationships include function calls, file inclusions, and variable declarations. The data model and the analysis is fixed for each tool, but the presence of a query language permits the tools to answer complex queries unanticipated by the original tool designers.

3. GRASPR

GRASPR [62] is an example of a cliche recognition system. Its analyzer builds a dataflow graph annotated with control flow information and other constraints. This abstract representation of programs and cliches allows **GRASPR** to efficiently deal with variations that less sophisticated techniques cannot handle. The power of Grasper comes from its library of cliches and its ability to detect instances of them in programs using an efficient graph parsing algorithm. Cliches can either describe common programming knowledge such as how to sort a list or knowledge of an application domain such as particular ways to schedule events in simulation programs.

4. DESIRE

One of the most ambitious systems yet developed is the **DESIRE** tool from the Microelectronics and Computer Consortium [63]. **DESIRE**'s goal is design recovery, and it uses informal knowledge such as variable names and comments as well as more traditional formal analysis to build a hierarchy of concepts that together describe a program. A further uniqueness of DESIRE is the fact that the concepts are domain concepts rather than program concepts thus stressing knowledge related to *why* a program is as it is rather than *what* it is or *how* it does what it does. DESIRE also has an experimental feature that uses neural network technology to help in the recognition process.

5. Tango

Sometimes, the most difficult part of understanding a program is appreciating the nuances of its core algorithms. There is an active research area called *software visualization* [64] that attempts to aid this process by providing graphic tools to display abstract representations of the fundamental operations and underlying data structures of a program during its execution. For example, an array being sorted can be viewed in order to provide a dynamic depiction of its contents as they become more ordered. The **Tango** tool [65] provides an analyst with a relatively easy mechanism for producing such animations. The analyst can annotate a program at key points to specify operations that drive the creation and modification of geometric shapes on a display screen. In the sorting example, the array elements are represented as blocks whose heights corresponds to their values, and which move from position to position as the program exchanges values. Of course, this sort of animation is useful not only in understanding the execution of an existing program but also in teaching algorithmic concepts to beginning programmers.

VI. Examples of Applications of Program Comprehension

Program comprehension is useful for a variety of purposes. Many of these purposes relate to acquiring sufficient knowledge of a system to enable migrating or adapting or enhancing the system. Each of these tasks adds its own challenges to the already difficult task of understanding a system. This section provides glimpses of a variety of such applications of program comprehension.

A. Database Migration

Many older commercial software systems were originally developed in Cobol, and, when they used any database technology at all, it was probably aimed at a mainframe computing environment. Many others used Cobol “flat files” to hold the data and contained abundant code to provide services currently provided as part of a commercial database management system. Today, relational database technology dominates, and there is a strong movement toward distributed computation on networked workstations. To migrate these older programs requires determining the structure of their data, deciding on an appropriate new organization, ascertaining the operations performed on the data, and determining how best to obtain those services in the new environment[66]. Approaches to solving the database re-engineering problem can be found in Batini, Ceri, and Navathe’s book [67].

B. User Interface Migration

A similar, albeit less well understood, problem concerns adapting a software system that uses one form of user interface technology to another. This might be as simple as moving from one graphical user interface (GUI) widget set to another or as ambitious as adding a graphical interface to a batch program. Unlike database migration where a small set of well-defined data schemas exist, the user interface migration problem must deal with issues such as the large variety of commercial toolkits and the subtlties of matching “look and feel.” In order to re-engineer an application to use a new user interface, the old interface must be determined and extracted, a replacement designed, and the code updated to support the new approach. Little work has been done in this area. The reader is referred to [68], [69], and [70].

C. Objectification

The recent interest in object oriented languages has led to the desire to migrate existing legacy systems into object-oriented languages such as C++. This requires the detection of candidate objects in the existing code and the affiliated operations. A "good" object requires a small set of related data items and a limited number of operations accessing them. Once candidates are identified, the code must be remodularized and translated into the target language. Work in this area is reported in [71], [72] and [73].

D. Specification Extraction

Another recent trend has been in the direction of increased use of formal methods in software development. A formal specification reduces ambiguity and enables proof techniques that can ascertain such properties as correctness, security, and safety. A formal specification involves a statement in some mathematical notation, such as predicate logic, of the conditions under which a subprogram can correctly operate (its *preconditions*) and another statement describing its effect on program state (its *postconditions*). For straight-line code and conditionals, automatically constructing the pre- and postconditions is straightforward; loops, however, cause problems. Understanding a loop requires constructing a predicate that describes its typical operation. Automated techniques for this task do not currently exist. Basili and Mills [74] describe the manual use of specification extraction to help understand a program. [75] describes attempts to automate this process. The CICS project [76] in England used a related technique to support the complete re-engineering of a large transaction processing system.

E. Business Rule Extraction

Large software systems can be thought of as capital assets. Often, they are the only written description of how an enterprise conducts business. As the accuracy of the user- and system manuals degrade over time, the system itself becomes more and more the arbiter of existing business conduct. It then becomes necessary to read the code or run the program to get answers to fundamental questions. This activity has come to be called *business rule extraction*, and CASE tool vendors are beginning to support the process.

VII. Challenges

Although there has been much progress in the area of program understanding, there still remains much to be done. This section describes outstanding challenges to researchers in this field of study.

A. Scaling Up

Many program understanding tools are restricted in various ways. The restrictions relate to the sheer size of the programs with which they are capable of dealing, their ability to deal with multiple concurrent processes, or the relatively low-level representations they provide. For example, one commercial tool organizes the results of its analyses into a database indexed by the names of the various functions, procedures, and global data in the systems being analyzed. The tool is incapable of dealing with a legacy system organized into several processes because external names may be duplicated in two or more distinct processes. That is, the database does not support instances of the same name used in two independent processes.

As the size of the system under study grows larger, issues such as architecture begin to dominate questions concerning the details of programming. Existing tools are much better able to deal with the latter than the former. In fact, the whole area of software architecture has only recently come under examination by the research community [77].

Another implication of large systems is that more than one person will necessarily be involved in their analysis. Questions arise as to how knowledge will be shared by the analysts and how redundant work can be reduced or avoided. To date, there is no work reported on the application of collaboration technology to the problems of program understanding.

A final problem relates to the sheer volume of data generated by the analysis of large systems. While dataflow diagrams and structure charts are valuable when navigating a system containing a hundred or so procedures, they are of little help when looking at systems one or two orders of magnitude larger. Preliminary work on this problem is reported in [78].

B. Process Issues

Although it is well known that effective process management can significantly aid the production of software, the study of this area has primarily concerned the development of systems from their initial conception to first delivery and not with their analysis and reengineering. For example, the development managers for an existing system may be confronted with the decision of whether to continue maintenance on the system, to reengineer it, or to scrap it and rebuild from scratch. There is little available data or guidelines to help management with this decision. Some process models have been proposed, but little validation has been done [79]. The Software Technology Support Center at Hill Air Force Base is compiling project histories [80]. And the Software Engineering Institute is producing a *Best Practices Guidebook*. One interesting experience report that contains an actual cost analysis can be found in [81], and Boehm includes a chapter in his book *Software Engineering Economics* on software maintenance projects [82]. Finally, Rugaber and Doddapaneni discuss transition strategies for the particular case of reengineering mainframe management information systems to a distributed workstation environment [83].

C. Use of Domain Knowledge

As discussed above, most progress in automated program understanding has leveraged compiler technology. The implication of this is that we can often answer *what* and *how* questions but not *why* questions, the answers to which are essential to taking full advantage of legacy systems. *Why* questions relate program constructs to the problem the program is supposed to solve. These application problems are typically couched in terms of an application domain such as the analysis of seismological data to locate underground reservoirs of oil or the computation of the accelerated depreciation allowance for income tax return preparation.

In order to fully understand these programs, the application domain must be understood and, preferably, modeled. A *domain* is a problem area, and application domain analysis is currently a very active area of research. Typically, many application programs exist to solve the problems in a single domain. Arango and Prieto-Diaz [84] give the following prerequisites for the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain. Once recognized, a domain can be characterized by its vocabulary, common assumptions, architectural approach, and literature.

What role might a domain description play in reverse engineering a program? In general, a domain description can give the reverse engineer a set of expected constructs to look for in the code. These might be computer representations of real world objects like tax rate tables or deductions. Or they may be algorithms, such as the LIFO method of appraising inventories. Or they might be overall architectural schemes, such as a client-server architecture for implementing a transaction processing system.

Because a domain is broader than any single problem in it, there may be expectations engendered by the domain representation that are not found in a specific program (the inventory algorithm may not appear in a program to compute personal income taxes but might in a business tax program). Because a program is not always accurate or up-to-date, there may be things missing or incorrectly expressed in the program, despite contraindications in the domain representation. And, because a program is often used for more than one purpose, it may include components that do not appear at all in the domain representation, such as a checkbook balancing feature in an income tax package.

Nevertheless, a domain representation can establish expectations to be confirmed in a program. Furthermore, the objects in the domain representation are related to each other and organized in prototypical ways that may likewise be recognized in the program. Hence, a domain representation can act as a schema for controlling the reverse engineering process and a template for organizing its results. Incorporating such domain information into the program understanding process is essential to answering the all-important *why* questions.

Domain analysis has not yet been applied to the problem of program comprehension. One exception to this is the **DESIRE** system described previously. Another recent research project in this area is described by DeBaud, Moopen, and Rugaber [85].

D. Validation

The study of program comprehension is an emerging discipline. As such, no readily accepted infrastructure exists to validate proposed advances. For example, there exist no agreed upon benchmarks for comparing analysis tools. Moreover, there does not even exist a mechanism for integrating tools. If, for example, there were a standard language for describing commonly occurring programming patterns, then a repository could be built upon which researchers could rely. Of course, the final arbiter of the value of this research are software development practitioners, and here too there is no agreed upon standard of evaluation, such as the *Turing Test* [87] proposed as a measure of the success of natural language understanding programs. The program understanding research community and the commercial vendors need to address these problems before their work will be widely accepted.

VIII. Resources

Aside from the references provided by this article, the reader is referred to the following sources of information on the topics of program understanding and reverse engineering.

- Conferences and workshops: the *International Conference on Software Maintenance*, the *Workshop on Program Comprehension,* the *Working Conference on Reverse Engineering,* the *Symposium on Partial Evaluation and Program Manipulation,* and the *Reverse Engineering Forum*.
- Journals: the *Journal of Software Maintenance* and the *Reverse Engineering Newsletter* of the Committee on Reverse Engineering of the IEEE Computer Society. Additionally, mainstream journals in the software area such as *IEEE Transactions on Software Engineering*, *IEEE Software,* the *Communications of the ACM,* the *Journal of Automated Software Engineering,* and *Software-Practice and Experience* occasionally publish articles on program analysis and understanding.
- Books: a few books have been published in this area: Arnold's *Software Reengineering* [88], and *Tutorial on Software Restructuring* [49], *Software Reuse and Reverse Engineering,* edited by P. A. V. Hall [89], Zuylen's *The REDO Compendium of Reverse-Engineering for Software Maintenance* [90], and the *Empirical Studies of Programmers* series published by Ablex. The "Re-engineering Tool Report" from Hill Air Force base [80] and the Software Engineering Institute's planned *Best Bractices Guidebook* both collect experiences from actual reengineering projects.
- Theses: recent theses have been published in the area by Allemang [91], Callis [92], Griswold [93], Hartman [94], Letovsky [95], Ning [96], and Wills [43].

Finally, a World Wide Web page that contains pointers to work in the area can be found at URL http://www.cc.gatech.edu/reverse/.

IX. References

[1] Barry W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[2] B. Lientz, E. Swanson, and G. E. Tompkins. "Characteristics of Application Software Maintenance." *Communications of the ACM,* 21(6), June 1978.

[3] R. K. Fjeldstad and W. T. Hamlen. "Application Program Maintenance Study: Report to Our Respondents." *Proceedings GUIDE 48,* Philadelphia, PA, April 1983.

[4] Elliot J. Chikofsky and James H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software,* 7(1):13-17, January 1990.

[5] H. A. Smith and J. D. McKeen. “Re-engineering the Corporation: Where Does I.S. Fit In?” *Proceedings of the Twenty-sixth Annual Hawaii Conference on Systems Sciences. Volume III: Information Systems: DSS/Knowledge-based Systems,* Jay F. Nunamaker, Jr. and Ralph H. Sprague, Jr., Editors, IEEE Computer Society Press, 1993, pp.120-126.

[6] Edsger W. Dijkstra. “Go To Statement Considered Harmful.” *Communications of the ACM,* 11(3):147-148, November 1968.

[7] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. “The Interleaving Problem in Program Understanding.” *2nd Working Conference on Reverse Engineering,* Toronto, Ontario, Canada, July 14-16 1995, to appear.

[8] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. “Designing Documentation to Compensate for Delocalized Plans.” *Communications of the ACM,* 31(11):1259-1267, November 1988.

[9] G. D. Bergland. “A Guided Tour of Program Design Methodologies.” *IEEE Computer,* October 1981, pp. 13-37.

[10] L. A. Belady and M. M. Lehman. “Programming System Dynamics or the Meta-Dynamics of System in Maintenance and Growth.” Technical Report RC 3546, International Business Machine Corporation, September 17, 1971.

[11] F. J. Lukey. “Understanding and Debugging Programs.” *International Journal of Man-Machine Studies,* 12(2):189-202, February, 1980.

[12] E. Soloway and K. Ehrlich. “Empirical Studies of Programming Knowledge.” *IEEE Transactions on Software Engineering,* SE-10(5):595-609, September, 1984.

[13] R. Brooks. “Using a Behavioral Theory of Program Comprehension in Software Engineering.” *Proceedings of the Third International Conference on Software Engineering,* May 10-12, 1978, Atlanta Georgia, IEEE Computer Society, pp. 196-201.

[14] Stephen B. Ornburn and Spencer Rugaber. “Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs.” *Proceedings of the Conference on Software Maintenance,* Orlando, Florida, November 1992, pp. 32-40.

[15] A. von Meyrhauser and A. M. Vans. “Program Understanding - A Survey.” CS94-120, Department of Computer Science, Colorado State University, August 1994.

[16] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro. “Approaches to Program Comprehension.” *The Journal of Systems and Software*, Elsevier North Holland, 14:79-84, February 1991.

[17] Santanu Paul, Atul Prakash, Erich Buss, John Henshaw. “Theories and Techniques of Program Understanding.” TR-74.069, IBM Canada Laboratory, October 1991.

[18] Alex Quilici. “A Memory-Based Approach to Recognizing Programming Plans.” *Communications of the ACM,* 37(5):84-93, May 1994.

[19] Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Little Brown, 1980.

[20] Ben Shneiderman. “Exploratory Experiments in Programmer Behavior.” *International Journal of Computer and Information Sciences*, 5(2):123-143, June 1976.

[21] G. H. Okimoto. “The Effectiveness of Comments: A Pilot Study.” IBM Technical report # SDD 01.1347, July 27, 1970.

[22] Ben Shneiderman. “Measuring Computer Program Quality and Comprehension.” *International Journal of Man-Machine Studies*, 9, 1977.

[23] L. Weissman. “Psychological Complexity of Computer Programs: An Experimental Methodology.” *ACM Sigplan Notices*, 9, 1974.

[24] P. R. Newsted. “FORTRAN Program Comprehension as a Function of Documentation.” School of Business Administration, University of Wisconsin, Milwaukee, Wisconsin, 1973.

[25] Tom Love. *Relating Individual Differences in Computer Programming Performance to Human Information Processing Abilities*. Ph.D. Thesis, University of Washington, 1977.

[26] Ben 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.

[27] Ruven Brooks. “Towards a Theory of the Cognitive Processes in Computer Programming.” *International Journal of Man-Machine Studies*, 9(6):737-741, 1977.

[28] Ruven Brooks. “Towards a Theory of the Comprehension of Computer Programs”. *International Journal of Man-Machine Studies*, 18:543-554, 1983.

[29] Susan Wiedenbeck. “Processes in Computer Program Comprehension.” *Empirical Studies of Programmers*, Eliot Soloway and Sitharama Iyengar, editors, Ablex Publishing, Norwood, New Jersey, 1986, pp. 48-57.

[30] E. Soloway and K. Ehrlich. “Empirical Studies of Programming Knowledge.” *IEEE Transactions on Software Engineering*, SE-10(5):595-609, September, 1984. Also published in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[31] Eliot Soloway, Jeffrey Bonar, and Kate Ehrlich. “Cognitive Strategies and Looping Constructs: An Empirical Study.” *Communications of the ACM*, 26(11), November 1983.

[32] Stanley Letovsky. “Cognitive Processes in Program Comprehension.” *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, editors, Ablex Publishing Company, Norwood, New Jersey, 1986.

[33] S. Letovsky and E. Soloway. “Delocalized Plans and Program Comprehension.” *IEEE Software*, 3(3):41-49, May 1986.

[34] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. “Designing Documentation to Compensate for Delocalized Plans.” *Communications of the ACM*, 31(11):1259-1267, November 1988.

[35] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[36] John Levine, Tony Mason, and Doug Brown. *Lex and Yacc 2nd Edition*. O’Reilly & Associates, October 1992.

[37] J. W. Backus. “The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.” *Proceedings of the International Conference on Information Processing*, UNESCO, Paris, France, June 1959.

[38] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

[39] T. J. McCabe. “A Complexity Measure.” *IEEE Transactions on Software Engineering*, 2(4):308-320, December 1976.

[40] W. P. Stevens, G. J. Myers, and L. L. Constantine. “Structured Design.” *IBM Systems Journal*, 13(2):115-139, 1974.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and its Use in Optimization.” *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, July 1987.

[42] Mark Weiser. “Program Slicing.” *Proceedings of the 5th International Conference on Software Engineering*, San Diego, California, March 9-12, 1981, IEEE Computer Society, pp. 439-449.

[43] Linda Mary Wills. *Automated Program Recognition by Graph Parsing. T*R 1358, MIT Artificial Intelligence Laboratory, July 1992.

[44] D. S. Scott and C. Strachey. “Towards a Mathematical Semantics for Computer Languages.” in *Computers and Automata*, J. Fox, editor, Polytechnic Institute of Brooklyn Press, 1971, pp. 19-46.

[45] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints.” *Fourth Annual ACM Symposium on Principles of Programming Languages*, Los Angeles California, January 1977, pp. 238-252.

[46] Herbert Ritsch and Harry M. Sneed. “Reverse Engineering Programs via Dynamic Analysis.” *Proceedings of the First Working Conference on Reverse Engineering*, Baltimore, Maryland, May 21-23, 1993, pp. 192-201.

[47] Elaine J. Weyuker. “The Evaluation of Program-Based Software Test Date Adequacy Criteria.” *Communications of the ACM*, 31(6):668-675, June 1988.

[48] Frank G. Pagan. *Partial Computation and the Construction of Language Processors*. Prentice Hall, 1991.

[49] Robert S. Arnold. *Tutorial: Software Restructuring*. April, 1986, IEEE Computer Society.

[50] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. “A Fortran-to-C Converter.” Computing Science Technical Report No. 149, AT&T Bell Laboratories, May 16, 1990.

[51] R. C. Waters. “Program Translation via Abstraction and Reimplementation.” *IEEE Transactions on Software Engineering*, 14(8):1207-1228, August, 1988.

[52] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[53] Elliot Chikofsky. *Computer-Aided Software Engineering (CASE), 2nd Edition*. IEEE Computer Society Press, 1993.

[54] James Martin and Carma McClure. *Structured Techniques: The Basis for CASE, Revised Edition*. Prentice Hall, 1988.

[55] P. P. Chen. “The Entity-Relationship Model-Toward a Unified View of Data.” *ACM Transactions on Database Systems*, March 1976, pp. 9-36.

[56] D. S. Wile. “Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages.” *Program Specification and Transformation*, L. G. L. T. Meertens, editor, Elsevier North Holland, 1987, pp. 165-195.

[57] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. “Research on Knowledge-based Software Environments at Kestrel Institute.” *IEEE Transactions on Software Engineering*, November 1985.

[58] *Software Refinery Toolkit*. Reasoning Systems Incorporated, Palo Alto, California.

[59] Premkumar T. Devanbu. “GENOA - A Customizable, Language- and Front-End Independent Code Analyzer.” *Proceedings of the Fourteenth International Conference on Software Engineering*, Melbourne, Australia, May 1992, pp. 307-319.

[60] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. “The C Information Abstraction System.” *IEEE Transactions on Software Engineering*, 16(3):325-334, March 1990.

[61] Judith E. Grass and Yih-Farn Chen. “The C++ Information Abstractor.” *1990 USENIX Conference*, 1990, pp. 265-277.

[62] L. Wills. “Flexible Control for Program Recognition.” *Working Conference on Reverse Engineering*, Baltimore, MD, May 1993.

[63] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. “Program Understanding and the Concept Assignment Problem.” *Communications of the ACM*, 37(5):72-83, May 1994.

[64] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. “A Principled Taxonomy of Software Visualization.” *Journal of Visual Languages and Computing*, 4(3):211-266, September 1993.

[65] John T. Stasko. “TANGO: A Framework and System for Algorithm Animation.” *IEEE Computer*, 23(9):27-39, September 1990.

[66] William Premerlani and Michael Blaha. “An Approach for Reverse Engineering of Relational Databases.” *Communications of the ACM*, 37(5):42-49+, May 1994.

[67] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design - An Entity-Relationship Approach*. Benjamin Cummings, 1992.

[68] E. Merlo, J. F. Girard, K. Kontogiannis, P. Panangaden, and R. DeMori. “Reverse Engineering of User Interfaces.” *Proceedings of the Working Conference on Reverse Engineering*, IEEE Computer Society Press, Baltimore Maryland, May 21-23, 1993.

[69] Larry Van Sickle, Zheng Yang Liu, and Michael Ballantyne. “Recovering User Interface Specifications for Porting Transaction Processing Applications.” *2nd Workshop on Program Comprehension*, July 1993, Capri, Italy.

[70] Melody Moore. Spencer Rugaber, and Phil Seaver. “Knowledge-based User Interface Migration.” *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 72-79.

[71] C. L. Ong and W. T. Tsai. “Class and Object Extraction from Imperative Code.” *Journal of Object-Oriented Programming*, 6(1):58-68, March-April 1993.

[72] Sying-Syang Liu, Roger Ogano, and Norman Wilde. “The Object Finder: A Design Recovery Tools.” SERC-TR-46-F, University of Florida, Software Engineering Research Center, January 1991.

[73] A. Cimitile, M. Tortorella, and M. Munro. “Program Comprehension through the Identification of Abstract Data Types.” *Proceedings of the 3rd Workshop on Program Comprehension*, pp. 12-19, Washington, D.C., November 1994.

[74] V. R. Basili and H. D. Mills. “Understanding and Documenting Programs.” *IEEE Transactions on Software Engineering*, SE-8(3):270-283, May 1982.

[75] Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner. “Using Function Abstraction to Understand Program Behavior.” *IEEE Software*, 7(1):55-63, January 1990.

[76] Ian J. Hayes. “Applying Formal Specification to Software Development in Industry.” *IEEE Transactions on Software Engineering*, SE-11(2):169-178, February 1985.

[77] Gregory Abowd, Robert Allen, and David Garlan. “Using Style to Understand Descriptions of Software Architecture.” *Sigsoft*, December, 1993, ACM, pp. 9-20.

[78] Stephen G. Eick, Joseph L.Steffen, and Eric E. Sumner Jr. “Seesoft---A Tool for Visualizing Line Oriented Software Statistics.” *IEEE Transactions on Software Engineering*, November 1992, 18(11):957-968.

[79] Lowell Jay Arthur. *Software Evolution*. John Wiley & Sons, 1988.

[80] Chris Sittenauer, Mike Olsem, and Daren Murdock. “Re-engineering Tools Report.” Software Technology Support Center, Hill Air Force Base, May 1992.

[81] Reginald L. Hobbs, John R. Mitchell, Glenn E. Racine, and Richard Wassmath. “Re-engineering Old Production Systems: A Case Study of Systems Re-development and Evaluation of Success.” *Emerging Information Technologies for Competitive Advantage and Economic Development: Proceedings of the 1992 Information Resources Management Association International Conference*, May 1992, Harrisburg, Pennsylvania, pp. 29-37.

[82] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[83] Spencer Rugaber and Srinivas Doddapaneni. “The Transition of Application Programs From COBOL to a Fourth Generation Language.” *Conference on Software Maintenance - 93*, September 27-30, 1993, Montreal, Canada, pp. 61-70.

[84] Guillermo Arango and Ruben Prieto-Diaz. “Domain Analysis Concepts and Research Directions” in *Domain Analysis and Software Systems Modeling*, Ruben Prieto-Diaz and Guillermo Arango, editors, IEEE Computer Society Press, 1991.

[85] Jean-Marc DeBaud, Bijith M. Moopen, and Spencer Rugaber. “Domain Analysis and Reverse Engineering.” *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 326-335.

[86] A. Cimitile, M. Tortorella, and M. Munro. “Program Comprehension through the Identification of Abstract Data Types.” *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, D.C., November 1994, pp. 12-19.

[87] Alan Turing. “Computing Machinery and Intelligence.” *Computers and Thought*, E. Feigenbaum and J. Feldman, editors, McGraw-Hill, 1963, pp. 1-35.

[88] Robert S. Arnold. *Software Reengineering*. IEEE Computer Society, 1993.

[89] P. A. V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*. Chapman and Hall.

[90] Henk van Zuylen, editor. *The REDO Compendium of Reverse Engineering for Software Maintenance*. John Wiley, 1992.

[91] D. Allemang. *Understanding Programs as Devices*. Ohio State University, Ph.D. Thesis, 1990.

[92] F. W. Calliss. *Inter-Module Code Analysis Techniques for Software Maintenance*. Ph.D. Thesis, 1989, University of Durham, Computer Science.

[93] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. Thesis, TR 91-08-04, Department of Computer Science & Engineering, University of Washington, August 1991.

[94] John Hartmann. *Automatic Control Understanding for Natural Programs*. Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, May 1991.

[95] S. Letovsky. *Plan Analysis of Programs*. Ph.D. Thesis, Yale University, 1988.

[96] J. Q. Ning. *A Knowledge-Based Approach to Automatic Program Analysis*. Ph.D. Thesis, University of Illinois at Urbana-Campaign, October 1989.