Program understanding: Challenge for the 1990s

by T. A. Corbi

In the Program Understanding Project at IBM's Research Division, work began in late 1986 on tools which could help programmers in two key areas: static analysis (reading the code) and dynamic analysis (running the code). The work is reported in the companion papers by Cleveland and by Pazel in this issue. The history and background which motivated and which led to the start of this research on tools to assist programmers in understanding existing program code is reported here.

"If the poor workman hates his tools, the good workman hates poor tools. The work of the workingman is, in a sense, defined by his tool—witness the way in which the tool is so often taken to symbolize the worker: the tri-square for the carpenter, the trowel for the mason, the transit for the surveyor, the camera for the photographer, the hammer for the laborer, and the sickle for the farmer.

"Working with defective or poorly designed tools, even the finest craftsman is reduced to producing inferior work, and is thereby reduced to being an inferior craftsman. No craftsman, if he aspires to the highest work in his profession, will accept such tools; and no employer, if he appreciates the quality of work, will ask the craftsman to accept them."[1]

Today a variety of motivators are causing corporations to invest in software tools to increase software productivity, including: (1) increased demand for software, (2) limited supply of software engineers, (3) rising expectations of support from software engineers, and (4) reduced hardware costs.[2] A key motivator for software tools in the 1990s will be the result of having software evolve over the previous decades from several-thousand-line, sequential programming systems into multimillion-line, multitasking "business-critical" systems. As the programming systems written in the 1960s and 1970s continue to mature, the focus for software tools will shift from tools that help develop new programming systems to tools that help us understand and enhance aging programming systems.

In the 1970s, the work of Belady and Lehman[3–5] strongly suggested that *all* large programs would undergo significant change during the in-service phase of their life cycle, regardless of the *a priori* intentions of the organization. Clearly, they were right. As an industry, we have continued to grow and change our large software systems to:

- Remove defects
- Address new requirements
- Improve design and/or performance
- Interface to new programs
- Adjust to changes in data structures or formats
- Exploit new hardware and software features

As we extended the lifetimes of our systems by continuing to modify and enhance them, we also

increased our already significant data processing investments in them and continued to increase our reliance on them. Software systems have grown to be significant assets in many companies.

However, as we introduce changes and enhancements into our maturing systems, the structure of the systems begins to deteriorate. Modifications alter originally "clean" designs. Fix is made upon fix. Data structures are altered. Members of the original and intervening programming teams disperse. Once current documentation gradually becomes outdated. System erosion takes its toll and key systems steadily become less and less maintainable, being more error prone and increasingly difficult and expensive to modify.

Flaherty's study[6] indicates the effect on productivity of modifying product code as compared to producing new code. His data for the studied System/370 communications, control, and language software show that productivity differences were greater between the ratio of changed source code to total amount of code than productivity differences between the different kinds of product classes—productivity was lowest when changing less than 20 percent of the total code in each product studied. The kind of software seemed to be a less-important factor contributing to lower productivity than did the attribute of changing a small percentage of the total source code of the product.

Clearly, as systems grow older, larger, and more complex, the challenges which will face tomorrow's programming community will be even more difficult than those of today. Even the *Wall Street Journal* stereotypes today's "beeper-carrying" programmer who answers the call when catastrophe strikes:

"He is so vital because the computer software he maintains keeps blowing up, threatening to keep paychecks from being issued or invoices from being mailed. He must repeatedly ride to the rescue night and day because the software, altered repeatedly over the years, has become brittle. Programming problems have simply gotten out of hand.

"Corporate computer programmers, in fact, now spend 80 percent of their time just repairing the software and updating it to keep it running. Developing new applications in this patchwork quilt has become so muddled that many companies can't figure out where all the money is going."[7]

Widespread routinization of computer programming and deskilling and fragmentation of programming work predicted by Kraft[8] has not occurred in the West because of management practices, the introduction of structured programming, and software production processes. To the contrary, the skills needed to do today's programming job have become much more diverse. To successfully modify some aging programs, programmers have become part historian, part detective, and part clairvoyant. Why?

Programmers have become part historian, part detective, and part clairvoyant.

"Software renewal" or "enhancement" programming is quite different from the kind of idealized software engineering programming taught in university courses as stated by Jones:

"The major difference between new development and enhancement work is the enormous impact that the base system has on key activities. For example, while a new system might start with exploring users' requirements and then move into design, an enhancements project will often force the users' requirements to fit into existing data and structural constraints, and much of the design effort will be devoted to exploring the current programs to find out how and where new features can be added and what their impact will be on existing functions.

"The task of making functional enhancements to existing systems can be likened to the architectural work of adding a new room to an existing building. The design will be severely constrained by the existing structure, and both the architect and the builders must take care not to weaken the existing structure when the additions are made. Although the costs of the new room usually will be lower than the costs of constructing an entirely new building, the costs per square foot may be much higher because of the need

to remove existing walls, reroute plumbing and electrical circuits and take special care to avoid disrupting the current site."[9]

The industry is becoming increasingly mired in these kinds of application software "renovation" and maintenance problems. Parikh[10] reports the magnitude of the problem through:

- Results of a survey of 149 managers of Multiple Virtual Storage (MVS) installations with programming staffs ranging from 25–800 programmers indicating that maintenance tasks (program fixes/modifications) represent from 55 to 95 percent of their workload
- Estimates that $30 billion is spent each year on maintenance ($10 billion in the United States) with 50 percent of the data processing budgets of most companies going to maintenance and that 50–80 percent of the time of an estimated one million programmers or programming managers is spent on maintenance
- A Massachusetts Institute of Technology study which indicates that for every $1 allocated for a new development project, $9 will be spent on maintenance for the life cycle of the project

Whereas today's modern design techniques and notations and wider acceptance of reusable software parts may help prevent propagating "old code" to future generations,[11] programmers will need tools to assist in reconstructing and analyzing information in previously developed and modified programs to aid them in debugging, enhancing, modifying, and/or rewriting "old" programs until these approaches take widespread hold in our critical systems.

"Software renewal" tools are needed to reduce the costs of modifying and maintaining large programming systems, to improve our understanding of programs so that we can continue to extend their life and restructure them as needed, and to build bridges from old software to updated software that is improved with new design techniques and notations and reuse technologies.

Just as library and configuration control systems were developed when the volumes of source code and the numbers of programmers working on a system increased, it is inevitable that new tools systems for managing the information about large programming systems will emerge to support long-term software renewal.

Approaches to aging systems

The notion of providing tools for program understanding is not new. Work in the 1970s,[12–16] which grew out of program proving, automatic programming and debugging, and artificial intelligence (AI) efforts, first broached the subject. Researchers stressed how rich program descriptions (assertions, invariants, etc.) could automate error detection and debugging. The difficulty of modeling interesting problem domains and representing programming knowledge, coupled with the problems of symbolic execution, has inhibited progress. Although there has been some limited success,[17] the lack of fully implemented, robust systems capable of "understanding" and/or debugging a wide range of programs underscores the difficulty of the problem and the shortcomings of these AI-based approaches.

Positive effects can result from restructuring.

Recognizing the growing "old program" problem present in mature applications, entrepreneurs have transformed this problem into a business opportunity and are marketing code-restructuring tools. A variety of restructuring tools have emerged (see Reference 18 for an examination of restructuring). The restructuring approach to address "old" programs has had mixed success. Although helpful in some cases for cleaning up some modules, restructuring does not appear to help in other cases.

One government study[19] has shown that positive effects can result from restructuring, including some reduced maintenance and testing time, more consistency of style, reduced violations of local coding and structure standards, better learning, and additional structural documentation output from restructuring tools. However, on the negative side, the initial source may not be able to be successfully processed by some restructurers that require modification before restructuring; compile times, load module size, and execution time for the restructured

program can increase; human intervention may be required to provide meaningful names for structures introduced by the tool.

Movement and replacement of block commentary is problematic for some restructurers. And, as has been observed, overall system control and data structures that have eroded over time are not addressed, as indicated by Wendel:

Automatically recapturing a design from source code is not considered feasible.

"If you pass an unstructured, unmodular mess through one of these restructuring systems, you end up with at best, a structured, unmodular mess. I personally feel modularity is more important than structured code; I have an easier time dealing with programs with a bunch of GO TOs than one with its control logic spread out over the entire program."[20]

In general, automatically recapturing a design from source code, at the present state of the art, is not considered feasible. But some work is underway and some success has been reported. Sneed et al.[21,22] have been working with a unique set of COBOL tools which can be used to assist in rediscovering information about old code via static analysis, to interactively assist in remodularizing and then restructuring, and finally to generate a new source code representation of the original software. Also, research carried out jointly by CRIAI (Consorzio Campano di Ricerca per l'Informatica e l'Automazione Industriale) and DIS (Dipartimento di Informatica e Sistemistica at the University of Naples) reports that the automatic generation of low-level Jackson or Warnier/Orr documents is totally consistent with COBOL source code.[23]

Both Sneed and CRIAI/DIS agree, however, that determining higher-level design abstractions will require additional knowledge outside of that which can be analyzed directly from the source code.

The experience of IBM's former Federal Systems Division with the aging Federal Aviation Administration's National Airspace System (NAS)[24] seems to indicate that the best way out is to relearn the old software, relying primarily on the source code, to rediscover the module and data structure design, and to use a structured approach[25–27] of formally recording the design in a design language which supports data typing, abstract types, control structures, and data abstraction models.

This process often proved to be iterative (from very detailed design levels to more abstract), but it resulted in a uniform means for understanding and communicating about the original design. The function and state machine models then provided the designer a specification from which, subsequently, to make changes to the source code.

The need to expand "traditional" software engineering techniques to encompass reverse engineering design and to address "software redevelopment" has been recognized elsewhere:

"The principal technical activity of software engineering is moving toward something akin to 'software redevelopment.' Software redevelopment means taking an existing software description (e.g., as expressed in a programming or very high-level language) and transforming it into an efficient, easier-to-maintain realization portable across local computing environments. This redevelopment technology would ideally be applicable to both (1) rapidly assembled system prototypes into production quality systems, and (2) old procrustean software developed 3 to 20 years ago still in use and embedded in ongoing organization routines but increasingly difficult to maintain."[28]

Definitions

Two working definitions are needed before discussing how program understanding relates to software renewal.

First, what is "old" code? It may be the manifestation of age that makes code old. Oldness may come from the lack of familiarity of the current programming team with the part of the system being enhanced. Modern programming practices now accepted as standard may not have been used "way back then" when the code was originally developed. Other key characteristics of old code, which are not necessarily linked to age, are poor design, a constraining design

point, use of an obsolete programming language, and/or missing or inaccurate documentation.

Was old code written a week ago or a decade ago? Unfortunately, the answer is that it could be either. Old code is existing code that cannot be easily understood, redesigned, modified, debugged, or rewritten. Why? It has the following attributes:

- Design was done with methods and techniques that do not clearly communicate the program structure, data abstractions, and function abstractions.
- Code was written with a programming language and techniques that do not quickly and clearly communicate the program structure, the program interfaces, data structures and types, and functions of the system.
- Documentation is nonexistent, incomplete, or not current.
- Design and code are not organized in such a way as to be insulated from changing external hardware or software.
- Design was targeted to system constraints that no longer exist.
- Code contains parts where nonstandard or unorthodox coding techniques were used.

Next, what are programmers doing when they are working on old code? The process of working on old code has acquired many names: software renewal, software evolution, program redevelopment, software renovation, "unprogramming," reverse engineering, and software maintenance. I have used the term *software renewal* here because for me that phrase carries more of the notion of enhancement. Today, however, *software maintenance* is still the term most commonly used to describe the process of working on old code, but it has a much wider connotation than just "fixing bugs." Parikh and Zvegintzov define the software maintenance process very broadly:

"... understanding and documenting existing systems; extending existing functions; adding new functions; finding and correcting bugs; answering questions for users and operations staff; rewriting, restructuring, converting, and purging software; managing the software of an operational system, and many other activities that go into running a successful software system."[29]

As defined by the National Bureau of Standards (NBS), "Software maintenance is the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production."[30] The NBS and others[31–33] generally recognize software maintenance as involving four major kinds of work:

To understand a program, three actions can be taken.

1. Corrective maintenance (20 percent), which acts to correct errors that are uncovered after software is in use, including diagnosis and fixing design, logic, or coding errors
2. Adaptive maintenance (25 percent), which is applied when changes in the external environment precipitate modifications to the software, such as new hardware, operating system changes, peripheral upgrades, etc.
3. Perfective maintenance (50 percent or more), which incorporates enhancements that are requested by the user community, such as changes, insertions, deletions, and modifications
4. Preventive maintenance (5 percent), which improves future maintainability and reliability and provides a basis for future enhancement

The above percentages are based on the Lientz and Swanson study[34] of 487 software development organizations and represent the distribution of the different kinds of software maintenance activities which those authors saw in the surveyed organizations. The numbers are consistent with the Fjeldstad and Hamlen survey[35] of 25 MVS, Virtual System 1 (VS1), and Disk Operating System (DOS) data processing installations and a government study[36] of software maintenance done by the U.S. General Accounting Office in 1981.

Understanding programs: A key activity

With software maintenance defined in this broad sense, studies indicate that "*more than half of the*

programmer's task is in understanding the system."[29] The Fjeldstad–Hamlen study[35] found that, in making an enhancement, maintenance programmers *studied the original program*

- About three-and-a-half times as long as they studied the documentation
- Just as long as they spent implementing the enhancement

In order to work with old code, today's programmers are forced to spend most of their time studying the only really accurate representation of the system.

To understand a program, three actions can be taken: read about it (e.g., read documentation); read it (e.g., read source code); or run it (e.g., watch execution, get trace data, examine dynamic storage, etc.). Documentation can be excellent or it can be misleading. Studying the dynamic behavior of an executing program can be very useful and can dramatically improve understanding by revealing program characteristics which cannot be assimilated from reading the source code alone. However, the source code is usually the primary source of information.

We all recognize that "understanding" a program is important, but most often it goes unmentioned as an explicit task in most programmer job or task descriptions. Why? The process of understanding a piece of code is not an explicit deliverable in a programming project. Sometimes a junior programmer will have an assignment to "learn this piece of code"—oddly, as if it were a one-time activity.

Experienced programmers who do enhancement programming realize, just as do architects and builders doing a major renovation, that they must repeatedly examine the actual existing structure. Old architectural designs and blueprints may be of some use, but to be certain that a modification will be successful, they must discover or rediscover and assemble detailed pieces of information by going to the site of the structure. In programming, this kind of investigation happens throughout the project:

- While requirements are being examined, lead designers or developers are typically navigating through the existing code base to get a rough idea of the size of the job, the areas of the system that will be impacted, and the knowledge and skills needed by the programming team which does the work.
- As design proceeds from high level to low level, each of the team members repeatedly examines the existing code base to discover how the new function can be grafted onto the existing data structures and into the general control flow and data flow of the existing system.
- Wise designers may "tour" the existing code to get an idea of performance implications that the enhancement may have on various critical paths through the existing system.
- Just before the coding begins, programmers are looking over the "neighborhood" of modules that will be involved in the enhancement. They are planning the detailed packaging—separating the low-level design into pieces which must be implemented by new modules or which can be fit into existing modules. Often, they are building the lists of new and changed modules and macros for the configuration management or library control team who need this information in order to reintegrate the new and changed source code when putting the pieces of the system back together again.
- During the coding phase, programmers are immersed in the old code. Programmers are constantly choosing between courses of action: making very detailed decisions to rewrite or restructure existing code versus decisions to change the existing code by deleting, moving, and adding a few lines here and a few lines there. Understanding the existing programs is also the key to adding new modules: How to interface to existing functions in the old code? How to use the existing data structures properly? How not to cause unwanted side effects?
- A new requirement or two and a few design changes usually come into focus after the programmers have begun their work. These additions must be evaluated as to their potential impact on the system and as to whether or not the proposed changes can be contained in the current schedules and resources. The "old base" and the "new evolving" code under development must be scrutinized to supplement the intuitions of the lead programmers before notifying management of the risks.
- Testers may delve into the code if they are using "white-box" techniques. Sometimes even a technical writer will venture into the source code to clarify something for a publication under revision.
- Debugging, dump reading, and trace analysis constantly require long terminal sessions of "program understanding" in which symptoms are used to postulate causes of an error, or bug. Each hypothesis causes the programmer to explore the existing system to find the source of the bug. When the

problem is found, a more "bounded" exploration is usually required to gather the key information necessary to actually build the fix and insert yet another modification into the system.

Therefore, the program understanding process is a crucial subelement in achieving many of the project deliverables: sizings, high-level design, low-level design, build plan, actual code, debugged code, fixes, etc.

The investigation process which programmers undertake when doing software maintenance is akin to idea processing.

Programmers attempt to understand a programming system so that they can make informed decisions about the changes they are making. The literature refers to this "understanding process" as "program comprehension":

"The program comprehension task is a critical one because it is a subtask of debugging, modification, and learning. The programmer is given a program and is asked to study it. We conjecture that the programmer, with the aid of his or her syntactic knowledge of the language, constructs a multileveled internal semantic structure to represent the program. At the highest level the programmer should develop an understanding of what the program does: for example, this program sorts an input tape containing fixed-length records, prints a word frequency dictionary, or parses an arithmetic expression. This high-level comprehension may be accomplished even if low-level details are not fully understood. At lower semantic levels the programmer may recognize familiar sequences of statements or algorithms. Similarly, the programmer may comprehend low-level details without recognizing the overall pattern of operation. The central contention is that programmers develop an internal semantic structure to represent the syntax of the program, but they do not memorize or comprehend the program in a line-by-line form based on syntax."[37]

The investigation process which programmers undertake when doing software maintenance is akin to *idea processing*, very clearly described by Halasz, Moran, and Trigg:

"The goal of all idea processing tasks is to move from a chaotic collection of unrelated ideas to an integrated, orderly interpretation of the ideas and their interconnections. Analyzing one's business competitors is a prototypical example. The task begins with an analyst extracting scraps of information about competitors from available sources. The collected information must be organized and filed away for subsequent use. More importantly, the collected information needs to be analyzed. The relationships between the various ideas have to be discovered and represented. Multiple analyses should be developed in order to understand the significance of the collected information. Once these analyses are complete, the analyst composes and writes a document or presentation that communicates the discovered information and its significance.

"Idea processing is a convolution of several different activities that can be roughly divided into three phases: acquisition, analysis, and exposition. Acquisition involves the capture or extraction of ideas and information from sources of various sorts, e.g., taking notes from a document or recording the ideas produced during brainstorming. Analysis involves discovering the significance of ideas, in particular, discovering the connections and relationships among ideas. Developing legal arguments based on case research is an example. Exposition involves communicating ideas and analyses in the form of reports, talks, etc."[38]

How do programmers learn to acquire key pieces of information about code, to organize and analyze it, and then to use it to make decisions?

Learning to understand programs

Although software engineering (e.g., applied computer science) appears as a course offering in many university and college computer science departments, software renewal, program comprehension, or enhancement programming are absent. In terms of the skills that are needed as our software assets grow and age, lack of academic training in how to go about understanding programs will be a major inhibitor to programmer productivity in the 1990s:

"... Unfortunately, a review by the author of more than 50 books on programming methodologies revealed almost no citations dealing with the productivity of functional enhancements, except a few minor observations in the context of maintenance.

"The work of functional enhancements to existing software systems is underreported in the software engineering curriculums, too, and very few courses exist in which this kind of programming is even discussed, much less taught effectively."[9]

For other "language" disciplines, classical training includes learning to speak, read, and write. Reading comprehension is a partner with composition and rhetoric. In school, we are required to read and critique various authors. An English education curriculum does not teach basic language skills (programming language syntax and semantics), recommended sentence structures (structured programming), and short stories (algorithms), expecting students to be fully trained, productive copy editors or authors for major publications upon completing the curriculum. Yet, many computer science departments sincerely believe that they are preparing their students to be ready for the workplace.

Unfortunately, most new college graduates entering today's software industry must confront a very considerable learning curve about an existing system before they get to the point where they can begin to try to do design or coding. They have little or no training nor much tool assistance to do this. Acquiring programming comprehension skills has been left largely to on-the-job training while trying to learn about an existing system.[39] Even experienced programmers can have trouble moving to a different project.

The lack of training and tools to help in understanding large, old programming systems also has another negative effect on productivity. It is resulting in a kind of job stagnation throughout the industry which Boehm terms the "Inverse Peter Principle":[40]

"*The Inverse Peter Principle:* 'People rise to an organizational position in which they become irreplaceable, and get stuck there forever.' This is most often encountered in software maintenance, where a programmer becomes so uniquely expert on the inner complexities and operating rituals of a piece of software that the organization refuses to let the person work on anything else. The usual outcome is for the programmer to leave the organization entirely, leaving an even worse situation."

As a large programming system grows older and older, more and more talented programmers will "get stuck" in accordance with the Inverse Peter Principle. Getting stuck directly impacts attempts by management to maximize project productivity by assigning the most talented programmers to get the next job done. Therefore, a lack of program understanding, training, and tools is a productivity inhibitor for new programmers on a project as well as a career inhibitor for the key project "gurus." As our programming systems grow in age, size, and complexity, these problems will compound, becoming increasingly more acute.

Theories of program understanding

Before we can build tools or begin any training programs, we must go deeper into the question: How do programmers "understand" a system? There are varying cognitive theories as to how a programmer constructs a "multileveled internal semantic representation" which Shneiderman[37] has postulated. Studies have been performed with programmers, and three current theories appear in the literature. Each theory appears plausible:

1. *The "bottom up" or "chunking" theory*—By reading the code, a programmer essentially iteratively "abstracts" a higher-level understanding of the program by recognizing and then "naming" more and more of the program. This is described in the book edited by Curtis:[41]

   "A process called 'chunking' expands the capacity of the short-term mental workspace. In chunking, several items with similar or related attributes are bound together conceptually to form a unique item. For example, through experience and training programmers are able to build increasingly large chunks based on solution patterns which emerge frequently in solving problems. According to Michael Atwood and Rudy Ramsey (1978),[42] the lines of code in the program listing:

```
      SUM = 0
      DO 10 I = 1, N
      SUM = SUM + X(I)
10    CONTINUE
```

   would be fused by an experienced programmer into the chunk 'calculate the sum of array X.' The programmer can now think about working with an array sum, a single entity, rather than the six unique operators and seven unique operands in the four program statements above. When it is

necessary to deal with the procedural implementation, the programmer can call from long-term memory these four statements underlying the chunk 'array sum.'

"As programmers mature they observe more algorithmic patterns and build larger chunks. The scope of the concepts that programmers have been able to build into chunks provides one indication of their programming ability. The particular elements chunked together have important implications for educating programmers. Educational materials and exercises should be presented in a way that best allows programmers to build useful chunks."[41]

2. *The "top-down" theory*—This theory proposes that programmers use their own experience and repeatedly try to confirm their expectations on the basis of what they believe the design to be. If they are told that the program they will be working on is a "payroll" system, just hearing that phrase before looking at the code causes them to expect to see certain constructs in the code: a master employee file with names, employee numbers, and salary fields; a timecard or attendance-gathering process; a process for updating salary; a process for deleting or adding employees; various report processors; a check-printing process; various exception-handling mechanisms (vacation, sickness, etc.).

   Now, when they pick up the code, they look for where these elements occur and "fill in" their belief of what the design most probably is. If something is missing or is radically different from their expectations (e.g., master file sorted by date of hire), the "surprise" causes some new experience to be stored for the next encounter.

   "The major points of the theory can be summarized as follows.

   a. The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain.
   b. Comprehending a program involves reconstructing part or all of these mappings.
   c. The reconstruction process is expectation driven by the creation, confirmation, and refinement of hypothesis."[43]

3. *The "opportunistic" theory*—The "opportunistic" theory says that understanding is a mixture of "top-down" and "bottom-up" strategies. Letovsky[44] believes that understanding a program involves a knowledge base (which represents the expertise and background knowledge a programmer brings to the task), a mental model (which is an encoding of the programmer's current understanding of the program), and an assimilation process:

   ". . . to direct the understander to turn pages and aim his eyeballs in certain directions, to take in information from the program and documentation text, and to construct the mental model. If we assume that the complete mental model resembles a procedural net, we immediately have a space of possibilities for how the assimilation process constructs it. It could represent the bottom or implementation layer first and build the annotation from the bottom up by recognizing plans. This approach is taken in Brotsky[45] and Shrobe.[46] Alternatively, the understander could represent the specification first and develop possible implementations top down using a planner or automatic programmer, ultimately matching the possible implementations against the code. This approach is used in Johnson–Soloway[17] and Brooks.[43] Our position is that the human understander is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available . . ."[44]

Which approach to program understanding is correct? As in most attempts to explain human problem-solving behavior, the answer is not clear. Different programmers may be predisposed toward one approach versus another on the basis of their level of experience or familiarity with the code.

". . . The data suggest the representation of the expert is more abstract and contains more general information about what a program does, whereas the representation of the novice is more concrete and contains information about how the program functions . . .

"The results of these experiments do not suggest that expert programmers lost the ability to attend to the details of a program . . . Rather, they suggest that experts have learned that, during comprehension of this type of program, paying attention to the abstract elements of the program is more important than paying attention to low level details . . ."[47]

The same programmer may use different approaches, depending on the kind of task which he or she has been asked to accomplish with respect to understanding the code. Some tasks require more complete understanding, whereas others may involve only a cursory inspection of the code. The understander's mental model will change as the process of investigating a piece of code progresses. The result of any investigation may result in a model which is incorrect or incomplete (with uninvestigated parts of the system or vaguely understood pieces) or contains ambiguities (such as multiple conjectures about what the same piece of code might do).

Some tasks require more complete understanding; others may involve only a cursory inspection of the code.

Regardless of which approach is employed, good evidence indicates that the more systematic a programmer is in investigating a program and the more complete the information which is gathered, the more likely the programmer will be successful in performing modifications to that program:

"Understanding how a program is constructed and how it functions are important parts of the task of maintaining or enhancing a computer program. We have analyzed videotaped protocols of experienced programmers as they enhanced a personnel database program. Our analysis suggests that there are two strategies for program understanding, the *systematic* strategy and the *as-needed* strategy. The programmer using the systematic strategy traces data flow and control flow through the program in order to understand global program behavior. The programmer using the as-needed strategy focuses on local program behavior in order to localize study of the program. Our empirical data show that there is a strong relationship between using the systematic approach to acquire knowledge about the program and modifying the program successfully. Programmers who used the systematic approach to study the program constructed successful modifications. Programmers who used the systematic strategy gathered *knowledge about the causal interaction of the program's functional components.* Programmers who used the as-needed strategy did not gather such causal knowledge and therefore failed to detect interactions among components of the program."[48]

Although the different theories can give us some ideas as to the cognitive processes in action, the difficulty of comprehending how large, aging programs work is illustrated in the work of Letovsky and Soloway,[49] who identify the problems of recognizing "delocalized plans—that is, programming plans realized by lines scattered in different parts of the program." Their empirical studies show that often a programmer will base a program repair or enhancement on very localized knowledge and partial understanding of the program, and this proves to be error prone, especially when neither the program nor the documentation reveals that specific pieces of code interact with other pieces of code (or data) some "distance" away.

Many large software systems, which were originally written before the software engineering techniques of data encapsulation and information hiding and before programming languages with type enforcement, provide many such opportunities for introducing errors during enhancement because of "widely" delocalized plans which work against large, arbitrarily evolved "control block structures." These baroquely connected pieces of storage, which form the backbone of many major software systems, typically have little or no access control. So, a program provided with an anchor pointer can often traverse areas of the structure that were never intended to be part of the data scope of that program.

Directions for program understanding in the 1990s

At IBM's Research Division, the Program Understanding Project began work in late 1986 on tools that could help programmers in two key areas: static analysis (reading the code) and dynamic analysis (running the code). The work is reported in two papers in this issue: "A program understanding support environment" by Linore Cleveland[50] and "DS-Viewer—An interactive graphical data structure presentation facility" by Donald Pazel.[51]

Cleveland's work focuses on exploiting the workstation multiwindow presentation of static analysis data

to give programmers a new way of reading programs. The assumption was a world where there were no listings and that assemblers and compilers put all their internally collected information (symbol table, control flow, data flow, cross reference, etc.) into a structured format which could be accessed from a high-performance workstation.

Pazel's work focuses on exploiting the workstation multiwindow presentation of dynamic data to give programmers a new way of viewing and navigating the dynamic data structures of a program which has been stopped during execution. The approach was to modify a host-based debugger to interface to a workstation-based presentation tool through which the programmer could explore dynamic data during a debugging session.

Tools and training should encourage development of different systematic investigative approaches.

We conjecture that combining these static and dynamic data views into a unified tool can provide an experimental base which can be used to observe programmers engaged in complex program understanding tasks. By studying programmers using such a tool, we would hope to add functions which could support systematic investigations that would assist in the interactive rediscovery of design information.

Given the empirical studies elsewhere and what we are beginning to learn from PUNS (Program UNderstanding Support) and DS-Viewer to date, programmer training and tools to assist in understanding existing systems should try to remain neutral in the techniques or functions that are offered and should not favor or force the use of only one way of gathering information about programs. Training and tools should be able to support "top-down," "bottom-up," and "opportunistic" approaches and not impose one theory of investigation on the programmer. The programmer should choose the process and/or tool function which best fits his or her level of expertise and the circumstances of the current assigned task.

Tools and training should show or teach the programmer how to develop strategies which make use of various kinds of basic program information, e.g., control flow, data flow, data declarations and structures, dynamic executions (trace), cross reference, module interface, call graphs, and even documentation. Both should show or assist the programmer in combining these different kinds of information in ways which can support his or her building or confirming hypotheses about the system being investigated.

While remaining flexible, tools and training should encourage development of different systematic investigative approaches so that the programmer can judge the progress and completeness of his or her inquiry on the basis of available data. Obtaining information about the system under study should be made as easy as possible so as not to require excessive effort which might preclude investigating something that could be important to understanding the system.

Tools that require complex query commands should not be used for program understanding, since often the programmer will be diverted from his or her primary purpose and begin struggling with secondary issues such as compound query formation and syntax errors. A good deal of attention to the "human factors" of program understanding tools is required.

Acknowledgments

I would like to thank several people who have contributed to this work. First, Linore Cleveland and Don Pazel, without whose excellent prototyping work all this would have remained just "idea-ware." I would also like to thank Andy Heller and Dr. Daniel Abensour for introducing me to the Research Division; Dr. Bruce Shriver for introducing me to the academic and professional community; Dr. Abraham Peled for believing in the project; Dr. Ashok Malhotra, Bill Harrison, and Vincent Kruskal for their collegial support; George Radin for his advice and counsel; Pat Goldberg for various inspirations; and Dick Butler for understanding the importance of this work.

Cited references

1. Gerald M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York (1971).
2. Barry W. Boehm, Maria H. Penedo, E. Don Stuckle, Robert D. Williams, and Arthur B. Pyster, "A software development environment for improving productivity," *IEEE Computer* **17**, No. 6, 30–44 (June 1984).

3. L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal* **15,** No. 3, 225–252 (1976).
4. M. M. Lehman and F. H. Parr, "Program evolution and its impact on software engineering," *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, IEEE Society Press (October 1976), pp. 350–357.
5. M. M. Lehman, "Laws of evolution dynamics—Rules and tools for programming management," *Proceedings of the Infotech Conference on Why Software Projects Fail*, London (April 1978), pp. 11/1–11/25.
6. M. J. Flaherty, "Programming process measurement for the System/370," *IBM Systems Journal* **24,** No. 2, 172–173 (1985).
7. Paul B. Carroll, "Computer glitch: Patching up software occupies programmers and disables systems," *Wall Street Journal* (January 22, 1988), p. 1.
8. Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States*, Springer-Verlag, New York (1977).
9. Capers Jones, "How not to measure programming quality" *Computerworld* **XX,** No. 3, 82 (January 20, 1986).
10. Girish Parikh, "Making the immortal language work," *International Computer Programs Business Software Review* **7,** No. 2, 33 (April 1987).
11. Ronald A. Radice and Richard W. Phillips, *Software Engineering: An Industrial Approach, Volume I*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988), pp. 14–19.
12. I. P. Goldstein, "Summary of MYCROFT: A system for understanding simple picture programs," *Artificial Intelligence* **6,** No. 1, 249–288 (1975).
13. S. M. Katz and Z. Manna, "Toward automatic debugging of programs," *SIGPLAN Notices* **10,** No. 6, 143–155 (June 1975).
14. G. R. Ruth, "Intelligent program analysis," *Artificial Intelligence* **7,** No. 1, 65–87 (1976).
15. S. M. Katz and Z. Manna, "Logical analysis of programs," *Communications of the ACM* **19,** No. 4, 188–206 (April 1976).
16. F. J. Lukey, "Understanding and debugging programs," *International Journal of Man-Machine Studies* **12,** No. 2, 189–202 (1980).
17. W. L. Johnson and E. Soloway, "PROUST: Knowledge-based program understanding," *Proceedings of Seventh International Conference on Software Engineering*, Orlando, FL (March 1984), pp. 369–380.
18. Robert S. Arnold, Editor, *Tutorial on Software Restructuring*, IEEE Computer Society Press, Washington, DC (1986).
19. *Parallel Test and Evaluation of a Cobol Restructuring Tool*, U.S. General Accounting Office, Washington, DC (September 1987).
20. Irv Wendel, "Software tools of the Pleistocene," *Software Maintenance News* **4,** No. 10, 20 (October 1986).
21. H. M. Sneed, "Software renewal: A case study," *IEEE Software* **1,** No. 3, 56–63 (July 1984).
22. H. M. Sneed and G. Jandrasics, "Software recycling," *IEEE Conference on Software Maintenance*, Austin, TX (September 1987), pp. 82–90.
23. P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile, "Maintenance and reverse engineering: Low-level design documents production and improvement," *IEEE Conference on Software Maintenance*, Austin, TX (September 1987), pp. 91–100.
24. Robert N. Britcher and James J. Craig, "Using modern design practices to upgrade aging software systems," *IEEE Software* **3,** No. 3, 16–24 (May 1986).
25. A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," *Proceedings of COMPSAC '77* (1977), pp. 242–251.
26. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).
27. H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer, and R. E. Quinnan, "The management of software engineering," *IBM Systems Journal* **19,** No. 4, 414–477 (1980).
28. Walt Scacchi, "Managing software engineering projects: A social analysis," *IEEE Transactions on Software Engineering* **SE-10,** No. 1, 49–59 (January 1984).
29. Girish Parikh and Nicholas Zvegintzov, Editors, *Tutorial on Software Maintenance*, IEEE Computer Society Press, Silver Spring, MD (1983), p. ix.
30. Roger J. Martin and Wilma M. Osborne, *Guidance on Software Maintenance*, NBS Special Publication 500-106, Computer Science and Technology, U.S. Department of Commerce, National Bureau of Standards, Washington, DC (December 1983), p. 6.
31. E. B. Swanson, "The dimensions of maintenance," *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, IEEE Society Press (October 1976), pp. 492–497.
32. Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Company, Inc., New York (1982), pp. 322–341.
33. James Martin and Carma McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1983), p. 3ff.
34. E. B. Swanson and B. Lientz, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley Publishing Co., Reading, MA (1980).
35. R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," *Proceedings of GUIDE 48*, The Guide Corporation, Philadelphia (1979).
36. *Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged*, AFMD-81-25, U.S. General Accounting Office, Washington, DC (February 1981).
37. B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer and Information Sciences* **8,** No. 3, 219–238 (1979).
38. F. G. Halasz, T. P. Moran, and R. H. Trigg, "NoteCards in a nutshell," *Conference on Computer Human Interaction and Graphic Interfaces Proceedings*, Toronto (April 1987), pp. 45–52.
39. Carolyn Van Dyke, "Taking 'computer literacy' literally," *Communications of the ACM* **30,** No. 5, 366–374 (May 1987).
40. Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981), p. 671.
41. Bill Curtis, Editor, *Human Factors in Software Development*, IEEE Computer Society Press, Washington, DC (1985), p. 7.
42. M. E. Atwood and H. R. Ramsey, *Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging*, U.S. Army Research Institute of Behavioral and Social Sciences, Technical Report TR-78-A21, Alexandria, VA (1978).
43. R. Brooks, "Toward a theory of comprehension of computer programs," *International Journal of Man-Machine Studies* **18,** No. 6, 542–554 (1983).
44. Stanley Letovsky, "Cognitive processes in program comprehension," in E. Soloway and S. Iyengar, Editors, *Empirical Studies of Programmers*, Ablex Publishing Corporation, Norwood, NJ (1986), pp. 58–79.
45. Daniel C. Brotsky, *An Algorithm for Parsing Flow Graphs*, Master's thesis, Massachusetts Institute of Technology, Cam-

bridge, MA (March 1984).
46. H. Shrobe, *Dependency Directed Reasoning for Complex Program Understanding*, AI-TR-503, Massachusetts Institute of Technology Artificial Intelligence Lab, Cambridge, MA (1979).
47. B. Adelson, "When novices surpass experts: The difficulty of a task may increase with expertise," *Journal of Experimental Psychology, Learning, and Cognition* **10**, No. 3, 483–495 (1984).
48. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," in E. Soloway and S. Iyengar, Editors, *Empirical Studies of Programmers*, Ablex Publishing Corporation, Norwood, NJ (1986), pp. 80–98.
49. S. Letovsky and E. Soloway, "Delocalized plans and program comprehension," *IEEE Software* **3**, No. 3, 41–49 (May 1986).
50. L. Cleveland, "A program understanding support environment," *IBM Systems Journal* **28**, No. 2, 324–344 (1989, this issue).
51. D. Pazel, "DS-Viewer—An interactive graphical data structure presentation facility," *IBM Systems Journal* **28**, No. 2, 307–323 (1989, this issue).

Thomas A. Corbi *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Corbi joined the IBM Palo Alto Development Center as a programmer in 1974 after graduating from Yale University. He worked in the General Products Division on text-processing, data management, and database systems. He was a development manager for IMS/VS Fast Path at the IBM Santa Teresa Laboratory in San Jose, California. He joined the Research Division at the Thomas J. Watson Research Center in 1982 and managed the Program Understanding Project from 1986 to 1988. He was also co-program chair of the 1988 IEEE Conference on Software Maintenance (CSM-88). Mr. Corbi is now on assignment to the Data Systems Division, focusing on improving programmer productivity.

Reprint Order No. G321-5360.