Delocalized Plans and Program Comprehension

A maintainer's understanding can go awry when it is based on purely local clues. How can we spell out the intentions behind a piece of code?

Stanley Letovsky and Elliot Soloway, Yale University

A maintainer must develop an understanding of a program if he is to carry out maintenance on it; the more complete and correct the understanding, the more likely that the modifications will be correct. Typically, however, the maintainer is under pressure to carry out the modification as quickly as possible. In such circumstances, he often forms only a local, partial understanding of the program, focusing his attention on the portion of the code in which the repair or enhancement is to be made. When neither the program nor the documentation reveals that specific pieces of code interact with other pieces of code some distance away, the formation of a purely local understanding can lead to an inaccurate understanding of the program as a whole, which in turn can result in incorrect or inefficient program modifications.

The above observation is based on a study we carried out with professional programmers engaged in a program enhancement task. In particular, we found that the programmers had difficulty understanding code following *delocalized plans*—that is, programming plans realized by lines scattered in different parts of the program. In this article we analyze the reasons for comprehension failures in such situations, and we go on to suggest program documentation strategies that should aid maintainers in comprehending code implementing delocalized plans.

A cognitive analysis of program comprehension

The research described here is based on a model of the cognition involved in program understanding.[1] According to this model, the goal of program understanding is to recover the intentions behind the code. We use the term *goal* to denote intentions and the term *plan* to denote techniques for realizing intentions. A plan can be thought of as a rewrite rule that rewrites a goal to a set of subgoals during program synthesis; the process can be reversed to achieve program understanding. Psychological evidence that programmers have learned repertoires of plans has been presented in past studies.[2] This parallels findings in other areas of expertise, such as chess,[3] where experts have similarly been found to possess libraries of significant recurring patterns.

Artificial intelligence researchers have used plans to organize knowledge about programming in a manner consistent with the ways people talk about programs. Rich[4] has developed a formalism for notating plans suitable for synthesis and analysis. He defines plans as reusable patterns of data flow and control flow. Johnson and Soloway[5] have developed a computer-based tutor that uses goals and plans to analyze bug-laden novice code. The use of goals and plans in analysis allows the tutor to provide students with error analyses expressed in terms of their intentions.

Plans are conceptually distinct from algorithms and procedures. The essential property of plans is that they can be composed in complex ways. For example, plans can be abutted, interleaved, nested, merged, and so on.[6] Algorithms are compositions of plans. For example, a mergesort algorithm can be viewed as the composition of a plan for recursion on a binary tree, a plan for splitting a sequence in two, a plan for sorting pairs of numbers, and a plan for merging sorted lists. These plans are familiar cliches to most programmers, but one would not speak, for example, of an *algorithm* for recursion on a binary tree. Similarly, most programming languages do not allow procedures to be composed in a way that would allow the above plans to be coded separately and composed together into a mergesort.

In our model, program understanding is viewed as a process of recognizing plans in the code. The central claim of this article is that the likelihood of a reader's correctly recognizing a plan in a program decreases when the lines of code are spread out or delocalized in the text of the program instead of being closely grouped. If the lines of code implementing a plan are close together, readers tend to have no trouble recognizing the plan.

In the following Fortran code fragment, we see an example of a "running-total loop" plan, a plan used to sum numbers being read in:

```
        Total = 0
 10     READ(10, * , END = 100) New
        Total = Total + New
        . . .
        GOTO 10
 100    . . .
```

Because the lines of code in this plan are in close proximity, a reader can (1) spot the key line or two that indicate the plan (e.g., "Total = Total + New") and (2) integrate all the lines of the plan into a coherent unit in his mind.

By contrast, when the lines of code in a plan are distributed throughout a program, recognition of that plan becomes problematic. For example, assume that the following line appears in one part of a program:

An earlier version of this article was published under the title "Strategies for Documenting Delocalized Plans," in *Proceedings of the Conference on Software Maintenance*, Washington, DC, Nov. 1985.

0740-7459/86/0500/0041$01.00 © 1986 IEEE

```
IREC = 0
```

Assume also that the following line appears in some other place, textually remote but downstream in the program's flow of control:

```
IF (IREC .EQ. 0) WRITE(6, *)  ''Record not found.''
```

Here, although the reader of this article can readily discern the plan, a reader of the program, encountering the first line in isolation, would have less to go on. In principle he could suspend judgment about the purpose of the first line until encountering the second line, or perhaps he could skip ahead in the program to the second line when he encountered the first. In practice, however, programmers avoid both options, preferring to guess at the goal on the basis of clues available from easily accessed information sources such as local code context, program documentation, and meaningful identifiers in the code. Although this strategy often works, it can also introduce false assumptions into the programmer's understanding, which may carry over into modifications of the program and produce incorrect, inefficient, or redundant code.

Description of protocol experiment

We videotaped six professional programmers engaged in adding a new feature to an existing program. The subjects consisted of four expert-level program maintainers and two junior-level program maintainers. The former had between three and 20 years of professional programming experience; the latter, less than three years.

The subjects were instructed to talk freely as they performed the task. In addition, they were asked to tell us, whenever they looked at a piece of code or documentation, what they were looking at and why. Interviewer prodding kept them talking when they fell silent.

This technique, called the ''thinking-aloud protocol,''[7] suffers from certain drawbacks. Analyzing the protocols is extremely time-consuming. As a consequence, protocol studies are almost always based on a small number of subjects, with the concomitant risk that the subject pool is unusual in some way. Moreover, the data from protocols consist primarily of the subjects' verbal statements. These are not typically amenable to quantification, so interpretation is limited to finding sequences of behavior that can be explained in terms of some theory.

On the positive side, thinking-aloud protocols offer one of the few windows on the cognitive processes of human subjects performing complex tasks. For this reason, protocol studies are frequently used to investigate human problem-solving behavior.

We presented each subject with a Fortran 77 program that managed a small, interactive database of personnel information such as names, addresses, and phone numbers. Called the PDB, for ''personnel database,'' the program contained 15 routines and approximately 300 lines of code.

In addition to the code, we provided the following program documentation:

- *Overview:* a brief description of the functionality of the program.
- *Program routine descriptions:* a description of each routine in terms of its specific function and the roles of its parameters and local variables (see Figure 1).
- *Hierarchy chart:* a diagram showing the calling structure of the routines.
- *File description:* a description of the database file structure.
- *Sample session:* a trace of an interactive session with the PDB.

Our intention was to make the documentation of the PDB reflect generic standards for program documentation.

The subjects were given the following instruction:

> The code enhancement we are asking you to make would allow the user to restore a deleted record to the database during the same session in which it was deleted. The restored record would thus be returned to active status and be available for subsequent access.

Three of the six subjects completed the task in the allotted 90 minutes.

Examples of comprehension failures

Some examples from the protocols will show how comprehension can go awry when it is based on purely local clues, as is often the case. The examples fall into two categories. In the first pair of examples, the reader forms an incorrect but locally plausible conclusion about the function of a piece of code. In the second pair, the reader correctly infers the code's intention but does not notice its redundancy. In all four cases the failures can be attributed to a combination of two factors: (1) the subject's tendency to interpret a piece of the code in terms of its local textual context, and (2) the absence of any explicit documentation of the nonlocal interactions.

Example 1: number of changes. Our first example involves a subject in the midst of writing a RESTORE subroutine for the PDB, using the existing DELETE subroutine as a template. (The use of a closely related subroutine as a template for a new one was a frequently observed coding strategy in these protocols.) He encounters a line of code in DELETE which he does not understand:

```
NCHNGE = NCHNGE + 1
```

He examines the documentation for DELETE, which describes the role of the variable NCHNGE:

> NCHNGE: contains the number of changes to the database made during the current session.

He says, ''Is restoring a record a change? I guess so.'' and proceeds to implement it as such, copying the NCHNGE = NCHNGE + 1 line from DELETE to RESTORE unchanged.

Magnum

In fact, the reason the number of changes was being counted was to test, at the end of a session, whether there was any need to rewrite the database: a dirty-bit plan. The PDB main routine (which the subject was not looking at) contained the line:

```
IF (NCHNGE .NE. 0) CALL PUTDB( . . . )        *a*
```

where PUTDB is the routine that writes out the database. Deletion of a record *is* a change in this sense, and so DELETE increments the counter, but a RESTORE is the *undoing* of a change and should decrement the counter. This programmer's patch will therefore write the database out more often than necessary. The above line of code (i.e., *a*) is diagnostic of the dirty-bit plan, and had the subject seen it, he would have had no problem making the correct decision in his implementation. Unfortunately, it was textually remote from the focus of his attention, and so he fell back on the strategy of making up a plausible answer based on the information at hand.

Example 2: zeroing a pointer. This example involves the following code fragment from the subroutine DELETE. DELETE does not physically delete the record; it only marks a status field in the record to indicate to other routines that the record should be ignored.

```
      SUBROUTINE delete( IREC, . . . )
C     Make a record unavailable for later access
      . . .
      STATUS(IREC) = ''deleted''              *a*
      IREC = 0                                *b*
      RETURN
      END
```

Line *a* sets the status field. The function of the STATUS array was documented elsewhere, and the subject demonstrated an understanding of this line. The function of the next line was less obvious. The documentation of the variable IREC was

IREC: contains the index in the database of the current record, or 0 if no such record exists.

The actual purpose of line *b* was extremely obscure. Elsewhere in the program there was a routine that searched the database for a desired record, using the name field as the search key. That routine, as an optimization, first checked whether the record operated on by the previous command had the desired name, as would be the case if the user operated on the same record in two successive commands. If the previous record has the same name, the search routine would resurrect the pointer from the previous command and exit without search. The purpose of line *b* was to inhibit the optimization in the search routine during the processing of the next command, since in the original program it would be incorrect to do any further processing on a deleted record. Again, nothing in the local text of the program or in the documentation indicated that line of code was in any way related to the search optimization.

Of our six subjects, four reacted visibly to line *b*. (By ''visi-

SUBROUTINE: create(dbase, name, nrecs, irec, nchnge)

LANGUAGE: Fortran 77

PURPOSE: To create a new record in the personnel database. The user is prompted to enter the data for each field in the record.

VARIABLES:

		TYPE	DESCRIPTION
INPUT:	name	character (length 60)	Contains the name of the person whose record is to be created

OUTPUT: NONE

INPUT/OUTPUT:

	TYPE	DESCRIPTION
dbase	character array (200,7) (length 60)	Contains all the records in the personnel database; dbase(i,j) is the jth field in the ith record; there are 7 fields in each record.
nchnge	integer	Contains the number of changes that have taken place to the personnel database (dbase) in a particular session.
nrecs	integer	Contains the number of records in the personnel database.
irec	integer	Contains the location of the record in dbase that is the same as the one that is to be created, or 0 if no such record exists.

INTERNAL: NONE

SUBPROGRAMS CALLED: error2

Figure 1. Sample of documentation.

bly'' we mean that the videotape shows an overt reaction: the subject scratches his head and says ''Why did they do that?'' The absence of a visible reaction does not mean a subject is not confused, but the presence of a reaction tells us something interesting is going on. In this case two subjects showed no reactions; they simply scanned that piece of code and said nothing.) One subject was very confused by this line, so he traced the data flow forward through the code until he found what it accomplished. Another, more experienced, reader rationalized it immediately in terms of the documentation: since the record was being deleted, he reasoned, the pointer should be changed to indicate a nonexistent record. Actually, the 0 value in line *b* served an entirely different function from its use as a record-not-found indicator. The second subject's analysis, although plausible, was local and not integrated with his understanding of other, nonlocal but causally connected parts of the program. His failure to understand line *b* caused his modified version of the program to search the database unnecessarily whenever a RESTORE directly follows a DELETE of the same record.

Diagnosis and prescription: roles vs. goals. An interesting feature of examples 1 and 2 is that the subjects had documentation describing the problematic variables. This documentation was straightforward and reasonable; some of the subjects remarked that the program had better than average documentation. Yet the documentation failed to help them get the understanding they needed in the cases we have described.

We can explain the first failure by drawing a distinction between the *role* of a variable and its *goal*. The role of a variable is the identity of the datum for which that variable is home. This is what the documentation provided:

```
NCHNGE:  contains the number of changes to the
         database
```

The role is distinct from the goal of the variable, which was implicit and undocumented. The goal of NCHNGE—that is, the reason for keeping track of the number of changes to the database—is to decide whether to write out the database.

How can we fix the documentation to help prevent comprehension failures like the ones just described? For example 1 the answer is relatively straightforward: we explicitly specify the goal of the variable along with its role:

```
NCHNGE:
  Role:  contains the number of changes to the database so far
  Goal:  used to test whether database file needs to be updated
         at the end of a session
```

Here we explicitly document the goal of NCHNGE so that wherever a reader encounters a fragment of the delocalized plan, he has access to the explanation for that plan. We believe that this form of documentation would have prevented our subject's error in this example.

In example 1, whenever the variable NCHNGE is assigned a value, the goal of the assignment is to keep its value consistent with its role, which is to hold the number of changes to the database. This means that the variable's role alone explains all updates to it. In other words, we can think of the documentation for NCHNGE as equivalent to a series of in-line comments, one for each line that updates NCHNGE. Thus, a line like

```
NCHNGE = NCHNGE + 1
```

in the context of documentation that reads

```
NCHNGE:  contains the number of changes to the
         database
```

has the same impact on the reader's understanding as this commented code line:

```
C  The database has been changed:
C  bump the change counter
   NCHNGE = NCHNGE + 1
```

The in-line comment states explicitly what the reader would deduce from looking at the code line and the documentation. The reader's deduction relies on an assumption that the variable has been updated in order to keep its value consistent with its role. We call a variable update that is performed to maintain the correspondence between a variable's value and its role a *normal* update. In the PDB program all updates to NCHNGE are normal updates. This simplifies comprehension of these updates.

In example 2 the use of the variable IREC is more complex than the use of NCHNGE in example 1. The documentation provided the following:

```
IREC:  contains the index in the database of the current record
```

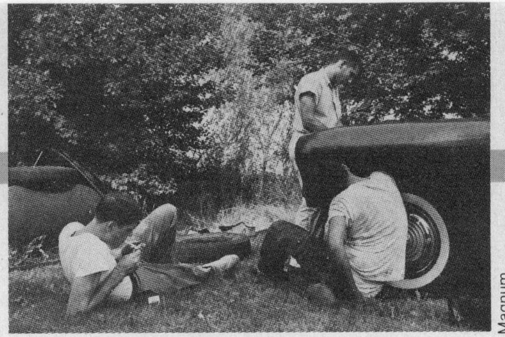
We cannot solve the comprehensibility problem by documenting the goal of IREC; indeed, doing so adds nothing:

```
IREC:
  Role:  holds pointer to the current record, or 0 if record not
         found
  Goal:  used to access the current record in the database
```

The problem in example 2 is not that the variable's role gave no clue about its goal, as in example 1, but rather that its role gave no clue about the purpose of some of its updates, particularly the one our subjects had trouble with. This update was a non-normal update: it changed IREC not because there was a new current record (which would have been a normal update), but rather to effect a secondary goal, the search optimization.

The strategy we prescribe for documenting more complex variables like IREC is to find their non-normal updates and document them with in-line comments. Normal updates need no comments, since we rely on the documentation of the variable's role and goal and on the conclusions the reader will naturally draw from them. For non-normal updates, we override the implications of the documentation with explicit in-line comments.

The normal updates to IREC occur at the beginning of a command, when a search for the record to be operated on is performed. The search ends by setting IREC to a new value. According to our strategy, these updates need no comments. The



updates that do need comments are those updating IREC in a manner unrelated to its role as pointer to the current record, including the update in example 2:

```
C  Zero IREC to inhibit SEARCH from
C  reusing record in next command
        IREC = 0
```

One could argue that a better way to address this problem is to forbid non-normal updates, to somehow force programmers to make all variables behave like NCHNGE. Indeed, various schools of normative programming styles (e.g., structured,[8] logic,[9] and functional[10]) derive much of their power by reducing or eliminating nonlocal interactions. We consider these valid approaches, but our concern here is effective documentation for code as we find it.

Example 3: caching previous value. This example concerns a plan for caching the previous values of a pair of loop variables from one loop iteration to the next. The old values are cached as part of an optimization whose goal is to prevent time-consuming searching of the database when the user operates on the same record twice in succession. The variables are NAME and IREC, which contain, respectively, the name of the person whose record is to be operated on by the current command, and the pointer to that person's record. The values of these two variables are saved at the end of each command in two other variables, OLDNAM and OLDREC (see *c*). During the next command, the database search routine checks whether the new name is the same as the name from the previous command (see *a*). If so, it restores IREC to its value from the previous iteration (see *b*) and proceeds without searching the database, as shown in this code fragment:

```
C    PDB Main Program
       . . .
C      Command loop: Perform DB commands
10
C        Loop body: Perform a command on a record
           . . .
C            Find pointer to current record
C              for this iteration, using name

C                Reuse pointer to previous record
C                if still correct
                 IF (NAME.EQ.OLDNAM) THEN          *a*
                   IREC = OLDREC                   *b*
                   GOTO 100
C                Else, search database for record
                 . . .
C              End of search, now have pointer in IREC
100            . . .
               OLDREC = IREC                       *c*
               OLDNAM = NAME
               GOTO 10
```

None of the programmers who studied this program in detail detected that line *b* has no effect. IREC is not changed between the fetch in the first command and the start of the next, so it is always already equal to OLDREC whenever the line is executed. This means that the variable OLDREC is also redundant. Moreover, so is the variable OLDNAM, since the name is available from the name field of the database using the pointer. (One could argue that writing things this way makes the code's meaning plainer and is therefore justifiable on readability grounds. But the failure of virtually all readers of the program to notice the statement's redundancy is significant. Comparable redundancies, when noticed, often upset our subjects greatly, prompting recommendations for rewriting.)

If we simplified the code fragment to eliminate this redundancy, we would have

```
C    PDB Main Program
       . . .
C      Command loop: Perform DB command on a record
10
C        Loop body: Perform a command on a record
           . . .
C            Find pointer to current record
C              for this iteration, using name

C                Reuse pointer to previous record
C                  if still correct
                 IF ( NAME .EQ. DBASE(IREC,1) )
                   THEN GOTO 100
C                Else, search database for record

C              End of search, now have pointer in IREC
100
               GOTO 10
```

The simplification eliminates three lines of code and two variables.

Example 4: overflow protection. This example concerns the manner in which the code achieves the goal of preventing database overflow. The database resides in an array during interactive sessions; this array holds at most MAXREC records. This use of an array gives rise to the goal of preventing any operations which would cause the program to operate on a database with more than MAXREC records and would thus overflow the array. The goal of preventing array overflow accounts for pieces of code in two subroutines: CREATE, the routine responsible for creating new records, and GETDB, the routine that reads the database into the array at the start of the session. In the following subroutines, these lines are marked with an *a*:

```
      SUBROUTINE create( . . . )
C     Create a new record in the database
        IF (NRECS .EQ. MAXREC) THEN                   *a*
          WRITE(6, *) ''Database full.''              *a*
          WRITE(6, *) ''Cannot perform command.''     *a*
          WRITE(6, *) ''Contact system manager.''     *a*
          RETURN                                      *a*
```

```
C   Else, really create record
        NRECS = NRECS + 1
        . . .
        RETURN
        END

        SUBROUTINE getdb( . . . )
C   Read the database into the array
        . . .
10      READ(13,20,END = 100) DBASE(IREC)
        IREC = IREC + 1
        IF(IREC .LT. MAXREC) GOTO 10                    *a*
        WRITE(6, *) "Database full."                    *a*
        WRITE(6, *) "Contact system manager."           *a*
        IERR = 1                                        *a*
        RETURN                                          *a*
100     . . .
        RETURN
        END
```

CREATE, the routine which adds new records to the database, compares the number of records NREC to the maximum allowable number MAXREC before allocating the space for a new record. If NREC is too large, an error message is printed and the allocation is not performed. In GETDB, the routine responsible for reading the database from a file into memory, there is a loop that reads in each record and keeps track of how many it has read using the variable IREC. Before it reads in each new record, it makes sure that IREC has not gotten larger than MAXREC. If IREC ever is larger, an error message is printed, and an error signal is returned to the calling routine in the variable IERR. Both fragments are locally plausible and seem to perform a reasonable task. Neither the original author of the program nor any of the readers noticed that since more than MAXREC records cannot be created, the file can never be larger than this, so the check in GETDB is redundant.

Diagnosis and prescription: plausible redundancies. While neither of these examples gave rise to any overt errors, we believe they illustrate an important problem for maintainers. Readers tend to

Maintenance has an irreversible quality: new code is added, but old code is seldom removed.

assume that each piece of code does something essential; this assumption is unlikely to be refuted in the course of maintenance carried out by maintainers who have an incomplete understanding of the code. Thus, maintenance has an irreversible quality: new code is added, but old code is seldom removed. The result is that the code increases in size and complexity, and becomes harder to understand with each successive modification.

The problem in examples 1 and 2 is in some sense the reverse of the problem in examples 3 and 4. In the former examples, there is difficulty understanding isolated lines of code in terms of the implicit plans that account for them. In the latter, the difficulty is finding all the lines of code that implement a plan. If that process were easier, it would be easier to see that the plans are redundantly implemented.

It is harder to design documentation to prevent the kind of problem seen in examples 3 and 4. First of all, we are not asking for quite the same thing as before. In the first two examples, we wanted documentation to prevent confusing code from being misinterpreted. But in examples 3 and 4, the original code was (unintentionally) redundant. We don't really want a strategy for documenting redundancy; what we would like is a documenting strategy that would help bring redundancy to the attention of the documenter, who could then eliminate it.

We believe that such a strategy requires a documentation entry for each plan in the program. Each entry would document the purpose and implementation of the plan and contain pointers to the lines of code that implement the plan. For instance, documenting example 3 in this fashion would give something like this*:

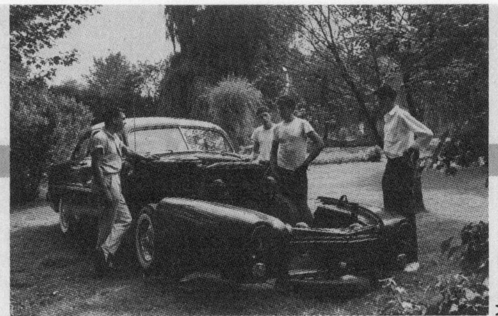
Goal: **Access Current Record**
  The current record is the record in the database whose name field matches the name entered by the user
Plan:
  (1) Get name from user into NAME
  (2) Store pointer to current record in IREC
  (3) Access current record using pointer in IREC

Goal: **Store Pointer To Current Record In IREC**
  Set IREC to index of record whose name field matches contents of NAME
Plan:
  (1) If NAME is the same as in previous command
      then reuse pointer from previous command
      else search DB for record

Goal: **Reuse Pointer From Previous Command**
Plan:
  (1) Store IREC in OLDREC after each command
  (2) Store OLDREC in IREC to reuse in next command

This example shows documentation for three of the program goals. Each entry states a goal together with the steps of the plan for achieving the goal. Each plan step, in turn, may be another goal. The first entry describes a plan whose goal is to make the current record accessible to some other part of the program. Its steps are to get the name, to store a pointer to the matching record in the variable IREC, and then to use the pointer to access the record. The second step is also a goal, shown here as the second entry. The goal statement is identical to plan step 2 of the first entry. The plan contains a single conditional step: if the name entered this time is the same as the name entered last time, then reuse the pointer from last time; otherwise, search the database.

*We are neglecting pointers to code, a complex presentation problem.


Magnum

The THEN case of the conditional is described in the third entry. It is here that the plan delocalized in the original code is presented in a coherent local grouping and its redundancy made more explicit.

Similarly, for example 4 a piece of documentation for the goal of preventing array overflow might look something like this:

Goal: **Prevent Overflow of Array**
Plan: Whenever NREC might be increased, guard against overflow
Places where NREC might be increased:
(1) when reading in db in GETDB
(2) when allocating new records in CREATE

The plan for achieving the goal is to insert code to guard against overflows in all places where NREC might be increased. By explicitly enumerating the places where this can happen, the documentation counters the delocalization of the plan and makes it more likely that the redundancy will be detected.

Implications and related work

The examples we have presented illustrate what we consider a fundamental problem for program maintenance: the tendency of programmers to make plausible but incorrect assumptions. We believe that programmers are most likely to believe an unverified assumption when they perceive that the effort required to verify the assumption is great, that the plausibility of the assumption is high, and that the importance of the assumption is small.

One way to eliminate the problem of incorrect assumptions is to forbid programmers from making assumptions at all. This is the approach taken by advocates of the most extreme form of structured programming,[8,11] in which programs are written together with in-line proofs of correctness. In this paradigm, a maintainer reads and understands the proof of correctness of the program while studying the code. We perceive several practical problems with this approach. First, the assertions included in a correctness proof are assertions about the states of the computation; according to our model, however, what the program reader is trying to determine is the intention behind the code. So a line of code such as IREC = 0 would be accompanied by the rather obvious assertion that X is now 0, but the reader would have no more clue than before as to why the line is there.

Second, the injunction that a programmer make no assumptions but reason clearly from explicit assertions is unlikely to carry much force. The process of making plausible assumptions about what is going on is basic to human cognition and is essential for such cognitive processes as understanding natural language and real-world stories (see Schank and Abelson[12]). To turn that process off in favor of rigorous application of deductive procedures goes against the grain, and requires considerable training. Indeed, advocates of this style of programming maintain that it must be ingrained from early on in the educational process to be learned at all. Moreover, since humans are unreliable calculators and programmers, they are likely to be unreliable deductive reasoners even at their best.

Finally, the no-assumptions injunction flies in the face of the cost assessments that seem to guide programmer behavior: it makes program understanding slower and more tedious, increasing the time involved in making a patch in exchange for a less immediately tangible increase in the probability of a correct enhancement.

Another approach to the problem is to provide the programmer with automatic program analysis tools that make the correct facts available to the programmer, reducing the need to make assumptions. Two tools of this kind are data flow analyzers and symbolic evaluators.[13] Data flow analyzers construct a graph of the possible data flow paths in a program; this graph is then presented to the programmer in some way. The information contained in such a graph is definitely relevant to the comprehension of delocalized plans; our first two examples can be described as erroneous assumptions about the ultimate use of a data value. Delocalized plans could be defined as plans with data flow links spanning widely separate parts of the code; data flow analyzers, by making such links explicit, could be very useful in countering the comprehensibility problems associated with these plans.

Symbolic evaluators perform automatically the kind of reasoning the structured programming advocates want programmers to do: they propagate assertions through the code, constructing a database that shows which facts are true at what points in the computation. Thus, they are potentially useful for preventing errors involving erroneous assumptions about the state of the computation. The redundancies in our second pair of examples could have been detected by symbolic evaluation.

For both types of tools, a major issue is the cost trade-offs involved in using the tool. There are two major types of overhead imposed by a tool. One is the mobilization cost: how much time and trouble is it to access the tool and apply it to the program? The other is the access cost: how easy is it to find the answer to a specific question with the tool? If the tool produces reams of paper analysis as output, programmers may prefer running the risk of an occasional incorrect guess to wading through the output in search of the right answer.

Another approach is to provide automatic program analysis tools that make the facts available, reducing the programmer's need to make assumptions.

Another tool worth mentioning here is the WEB system developed by Knuth,[14] which supports the development of programs and documentation together in a single document. The system

provides two compilers: one transforms the WEB source into a richly indexed, structured document that includes the pretty-printed source code of the program and arbitrary quantities of natural language exposition; the other generates a program suitable for compilation. The system allows the code to be taken apart into fragments, which can be presented in whatever order is convenient for exposition, while retaining the ability to reassemble a working program from the fragments. The WEB system speaks to the need for richly indexed program documentation and to the possibility that the presentation of the program for optimal human comprehensibility may not be the same as the presentation demanded by a compiler. Neither the system nor its author takes a position on how programs should be presented to optimize comprehensibility, however. Moreover, the reliance on paper documents as a medium of presentation imposes costs such as the need to look things up in the index, to find and search the pages listed under an index entry for the desired piece of information, and so on.

Another line of research with implications for improved program comprehension is the work on intelligent software development systems, such as the Programmer's Apprentice project at MIT,[15] the transformational programming group at the Informational Sciences Institute at USC,[16,17] and the automatic programming group at Kestrel Institute. These efforts involve the use of formal representations of plans and transformations to represent programs, specifications, and the relationships between them. Various testbeds are under development to explore the usefulness of these representations in rapid prototyping and program maintenance.

Our approach to the problem of program understanding relies on psychological methods for finding answers to the following pair of questions:

- What information needs to be provided to the reader?
- When and how should it be provided?

In this article we have advanced a restricted answer: information needed to form correct interpretations of delocalized plans should be easily accessible when components of those plans are encountered.

We are engaged in a long-range project to develop a computer-based tool to interactively supply information about the program to the reader. This tool will represent both the program and its specification in a plan formalism, from which it will generate responses to queries. Our approach to documentation takes the following stand on the above two questions:

- The reader needs to know the intentions behind the code.
- The cost of accessing information, in terms of time and trouble, must be made so low that readers will find it convenient to routinely verify their assumptions.

In other words, readers need to be able to get answers to their questions about programs easily. Our hope is that interactive presentation will support such access. We have recently performed an analysis of our protocol data to develop a taxonomy of the kinds of questions readers ask[1] to enable us to design an appropriate query interface.

We have presented examples from protocol studies of expert programmers, illustrating certain common kinds of comprehension errors that can occur in the reading of code during maintenance. These errors involve programming plans which are delocalized—that is, spread far and wide in the text of the program. Our studies suggest that delocalized plans are more liable to misinterpretation than plans whose code is closely grouped.

We have described the following strategies for preventing comprehension failures due to delocalization:

- Document variable goals as well as roles.
- Document non-normal updates with in-line comments.

We are planning to test these strategies empirically by investigating whether the recommended documentation facilitates better maintenance.

We have also provided examples to illustrate how delocalization of plans fosters redundant code. The solution to this involves moving from documenting the code itself to documenting the plans in the code. A fully satisfactory technique for documenting plans must await the design of suitable automated tools. □

Acknowledgment

Research described in this article was supported by and carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

References

1. S. Letovsky, "Cognitive Processes in Program Comprehension," to appear in *Proc. Conf. Empirical Studies of Programmers*, 1986.
2. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Engineering*, Vol. SE-10, No. 5, 1984, pp. 595-609.
3. A.D. deGroot, *Thought and Choice in Chess*, Mouton and Company, Paris, 1965.
4. C. Rich, "Inspection Methods in Programming," Technical Report AI-TR-604, MIT AI Lab, 1981.
5. W.L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *Proc. Seventh Int'l Conf. Software Engineering*, Orlando, Fla., IEEE Computer Society, 1983.
6. Richard C. Waters, "A Method for Analyzing Loop Programs," *IEEE Trans. Software Engineering*, Vol. SE-5, No. 3, 1979, pp. 237-247.
7. K.A. Ericsson and H.A. Simon, *Protocol Analysis: Verbal Reports as Data*, MIT Press, Cambridge, Mass., 1984.
8. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
9. K.L. Clark and S.A. Tarnlund, eds., *Logic Programming*, Academic Press, New York, 1982.
10. P. Henderson, *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1980.
11. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

12. R.C. Schank and R. Abelson, *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum, Hillsdale, N.J., 1977.
13. S.S. Muchnick and N.D. Jones, eds., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
14. D. Knuth, "Literate Programming," *Computer Journal*, Vol. 27, No. 2, May 1984, p. 97.
15. Richard C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," *IEEE Trans. Software Engineering*, Vol. SE-8, No. 1, 1982, pp. 1-12.
16. R. Balzer, "Transformational Implementation: An Example," *IEEE Trans. Software Engineering*, Vol. 7, No. 1, Jan. 1971.
17. R. Balzer, N. Goldman, and D. Wile, "On the Transformational Implementation Approach to Programming," *Proc. Second Int'l Conf. Software Engineering*, IEEE Computer Society, Long Beach, Calif., 1976.

Stanley Letovsky received his BA from Cornell University, where he was a College Scholar specializing in neurobiology and computer science. From 1979 to 1981 he was a research associate in the cardiovascular research laboratory at Harvard Medical School, where he developed software for analyzing analog measurements of cardiac function. Letovsky received his MS in computer science from Yale University in 1982. He is currently working at Yale on a PhD in artificial intelligence. His research interests include automatic programming and the psychology of programming.

Elliot Soloway is an associate professor in the Department of Computer Science at Yale University. He is also vice president at Compu-Teach, Inc., a New Haven company that produces educational software. In addition, he consults for a number of major software and AI companies. Soloway and his research group at Yale are exploring programming from an AI/cognitive science perspective, with particular interest in the implications of this work for software engineering and education.

Soloway has a BA in philosophy from Ohio State University and MS and PhD degrees in computer and information science from the University of Massachusetts in Amherst.

Readers may write to the authors at Yale University, Dept. of Computer Science, PO Box 2158, New Haven, CT 06520.

TOTAL CONTROL with *LMI FORTH*™

For Programming Professionals: an expanding family of compatible, high-performance, Forth-83 Standard compilers for microcomputers

For Development:
Interactive Forth-83 Interpreter/Compilers
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 400 page manual written in plain English
- Options include software floating point, arithmetic coprocessor support, symbolic debugger native code compilers, and graphics support

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8086, 68000, 6502, 8051, 8096, 1802, and 6303
- No license fee or royalty for compiled applications

For Speed: CForth Application Compiler
- Translates "high-level" Forth into in-line, optimized machine code
- Can generate ROMable code

Support Services for registered users:
- Technical Assistance Hotline
- Periodic newsletters and low-cost updates
- Bulletin Board System

Call or write for detailed product information and prices. Consulting and Educational Services available by special arrangement.

LMI *Laboratory Microsystems Incorporated*
Post Office Box 10430, Marina del Rey, CA 90295
Phone credit card orders to: ***(213) 306-7412***

Overseas Distributors.
Germany: Forth-Systeme Angelika Flesch, Titisee-Neustadt, 7651-1665
UK: System Science Ltd., London, 01-248 0962
France: Micro-Sigma S.A.R.L., Paris, (1) 42.65.95.16
Japan: Southern Pacific Ltd., Yokohama, 045-314-9514
Australia: Wave-onic Associates, Wilson, W.A., (09) 451-2946

Reader Service Number 6