Mental Models and Software Maintenance

David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway
Yale University, New Haven, Connecticut

Understanding how a program is constructed and how it functions are significant components of the task of maintaining or enhancing a computer program. We have analyzed vidoetaped protocols of experienced programmers as they enhanced a personnel data base 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 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 a 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 as-needed approach failed to construct successful modifications. Programmers who used the systematic strategy gathered *knowledge about the causal interactions 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.

1. INTRODUCTION

Understanding how a program is constructed and knowing how it functions are significant components of the task of maintaining or enhancing a computer program. In an early study of professional, expert maintenance programmers, Fjeldstad and Hamlen [1] discovered that the maintenance programmers they studied spent a major portion of their time "understanding the intent and style of *implementation* of the original programmer" (italics added). Fjeldstad and Hamlen found that, in making an enhancement, maintenance programmers studied the original program

- about 3.5 times as long as they studied the documentation of the program, and
- just as long as they spent implementing the enhancement.

In view of the observation that the professional maintenance programmers Fjeldstad and Hamlen studied spent as much time understanding the program as constructing the enhancement, it is clear that understanding the original program was very important to them.

What it means for an expert maintenance programmer to "understand" a computer program, however, it is not clear. Does it mean knowing all the variables used in the program? All the subroutines? All the data structures? How the program behaves when it executes? In order to determine what it means for expert maintenance programmers to understand a computer program, we studied professional maintenance programmers implementing an enhancement to an existing program. The results of our program enhancement study with maintenance programmers led to two intimately related conclusions.

1. There are two basic strategies for understanding computer programs.
2. The strategy a programmer uses to study the program strongly influences the programmer's understanding of the program.

Thus, the programmer, by using one of two strategies to study the program, develops one of two understandings of the program in one of two ways. We will show that one strategy for program understanding resulted in superior performance on the enhancement task.

In this paper, we examine the two strategies for program understanding, the systematic strategy and the as-needed strategy. The systematic strategy and the as-needed strategy stem from differences in the extent to

Address correspondence to David C. Littman, Cognition and Programming Project, Department of Computer Science, Yale University, New Haven, CT 06520.

The research reported in this paper was supported by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. This article was originally published in Soloway, E., and Iyengar, S. (eds.), Empirical Studies of Programmers, Ablex Publ. Corp., Norwood, N.J., 1986. Reprinted with permission from the publisher.

0164-1212/87/$3.50

which the programmers want to understand the program.

Systematic Strategy: The programmer using the systematic strategy wants to understand how the program behaves before attempting to modify it. To learn how the program behaves when it executes, this programmer performs extensive symbolic execution of the data flow paths between subroutines. By performing extensive symbolic execution of the data flow paths between subroutines, the programmer detects causal interactions among components of the program. Knowledge of the causal interactions in the program permits the programmer using the systematic strategy to design a modification that takes these interactions into account.

As-Needed Strategy: In contrast to the programmer using the systematic strategy, the programmer using the as-needed strategy attempts to minimize studying the program to be modified. This programmer attempts, as soon as possible, to localize parts of the program to which changes can be made that will implement the modification. Since the programmer using the as-needed strategy does not approach the modification task with a good understanding of the program when he or she attempts to implement a modification, it becomes necessary to gather additional information while the program modification is performed. The specific questions that arise as a result of attempting to modify the program determine what this programmer learns about the program. Thus, there is no guarantee that he or she will ever focus on the data flow among interacting subroutines. As a result, the programmer using the as-needed strategy is unlikely to uncover interactions in the program that might affect or be affected by the modification.

Thus, the systematic strategy and the as-needed strategy arise primarily from different goals. The programmer using the systematic strategy traces data 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 a 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 as-needed approach failed to construct successful modifications. We observed that programmers using the systematic strategy and those using the as-needed strategy acquired different knowledge about the program. In particular, programmers who used the systematic strategy gathered knowledge about the causal interactions 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.

In the rest of this paper, we explore the implications of using the systematic and as-needed strategies for understanding programs. We will show that there is a natural relationship between the use of a study strategy and the understanding of the program the programmer acquires. In Section 2 we will provide a description of our study and the enhancement task we gave professional programmers. In Section 3 we will discuss the knowledge required to construct a successful enhancement. Section 4 will introduce the concept of mental models representing the programmer's understanding of the program. In Section 5 we will present examples, drawn from our programmers, that illustrate the knowledge acquired through using the systematic and the as-needed strategies. Section 6 will report statistical data supporting the relationship between successful enhancement of the program and the use of the systematic approach to understanding the program. Finally, in Section 7, we will discuss some of the conclusions and implications of this study.

2. METHODOLOGY

2.1. Subjects

Ten professional programmers from Jet Propulsion Laboratory, California Institute of Technology, participated in this study. Each programmer had at least 5 years of programming experience. Subjects ranged in professional programming experience from 1 year to 22 years.

2.2. Data Collection

Each subject was interviewed individually. Interviews were recorded on videotape. During each interview, subjects were asked to describe their thoughts as they performed the task. The interviewer prompted the subject and asked questions to ensure that the subject produced a steady report of his or her thoughts.

2.3. A Maintenance Task

We presented subjects with a 250-line, 14-subroutine FORTRAN program that maintains a data base of personnel records of the employees of a small company. The personnel data base program (PDB) permits records to be created, updated, deleted, and displayed. The unmodified PDB program does not, however, allow a user to restore a record deleted during a session. The programmer's task, therefore, was to add a subroutine, called RESTORE, that permitted the user of the data base program to restore records deleted during a session through the use of the DELETE subroutine. Subjects attempted to solve the task presented in Figure 1.

3. KNOWLEDGE REQUIRED FOR ACCOMPLISHING THE TASK

In order to understand a program, a programmer must know about the *objects* the program manipulates and the *actions* the program performs. In addition, the programmers must have knowledge about the *functional components,* consisting of functionally related actions, that accomplish tasks in the program. Because these aspects of the program do not change as the program runs, but remain static, we call the programmer's knowledge of these objects, actions, and functional components *static knowledge.* A programmer must also know about causal connections between functional components in the program as the program runs. We call this knowledge about the interactions among functional components *causal knowledge.* In the next section, we outline the specific static and causal knowledge programmers needed to construct correctly a RESTORE enhancement.

3.1. Static Knowledge Required

To construct a successful RESTORE enhancement, a programmer needs at least four key pieces of static knowledge. The programmer must know about the objects the program manipulates:

1. Each personnel data base record contains a *record activity status field,* which may be given the value "active" or the value "deleted," depending on whether the personnel record is active in the data base or deleted from it.

The programmer must also know about three primary functional components of the PDB program:

2. In order to delete a record from the PDB, the DELETE subroutine changes the value in the activity status field in the record from active to deleted.
3. Records are never physically deleted from the PDB.
4. Since the record search subroutines bypass records marked "deleted," the record search process finds only active records.

ASSIGNMENT

The Personnel Data Base System provides on-line personnel information. Today, we ask you to increase the functional capability of the system by making the following enhancement:

Allow the user to restore a record that was deleted during the current session. For example, assume that the user deleted the following record during a session with the Personnel Data Base System:

Soloway, Elliot, M.
177 Howard Ave.
New Haven CT 06519
203 562-4151
Dunham Labs 322C
436-0606

Deleting a record makes that record unavailable for subsequent access. The enhancement we are asking you to make would allow the user to restore a deleted record to the data base during the same session that it was deleted in. For example, a user who had deleted the above record could then restore it during the same session. The record is thus returned to active status and is available for subsequent access.

Figure 1. The RESTORE enhancement task.

In short, without knowing that personnel records are deleted by changing the value in the record's activity status field from active to deleted, and that the search subroutines find only active records, the programmer cannot construct a successful RESTORE enhancement that introduces minimal modifications to the existing PDB program.

3.2. Causal Knowledge Required

Causal knowledge about the PDB program permits the programmer to reason about the interactions of the program's functional components during execution. An important aspect of the program's behavior is the way in which data flow permits structurally separate program components to *interact causally* during execution. In particular, in order to construct a successful RESTORE enhancement, the programmer must recognize the interaction between RESTORE's need for records marked "deleted" and the sequence in which the record search process and the transaction subroutines CREATE, DELETE, SHOW, and UPDATE are called: Since the record search process is carried out *before* calling the transaction subroutines, the same record search process must fulfill the precondition of finding the appropriate record for all transactions, including RESTORE. Unmodified, however, the record search process cannot provide active records for CREATE, DELETE, SHOW, and UPDATE and also provide deleted records for RESTORE. Therefore, if RESTORE uses the same record search process the other transaction subroutines use, then the search process must be modified to fulfill RESTORE's need for a deleted record. Unless the programmer knows how the data flow satisfies preconditions for the transaction subroutines, this interaction between RESTORE and the search routines will remain undetected and will hinder the construction of a successful RESTORE enhancement.

As programmers studied the PDB program, they gathered both static knowledge and causal knowledge about it. We call the programmer's knowledge about the PDB program a *mental model* of the program. In Section 4 we introduce the two strategies programmers used to study the program and the mental models they built as a result of each strategy.

3.3. Prototypical Solution for RESTORE Task

Programmers who correctly performed the RESTORE enhancement task attempted to respect both the PDB program's modularity and its calling hierarchy in their solutions. Respecting the modularity of the PDB program led successful programmers to refrain from adding new subroutines to perform the record search process when the user requested the RESTORE transaction; respecting the calling hierarchy caused successful programmers to call the record search subroutine at the same point in the program regardless of the record transaction requested by the user of the PDB program. To satisfy both constraints, successful programmers made two modifications to the search subroutine:

1. A parameter identifying the requested transaction was passed to the record search subroutines.
2. The actual search for records in the data base was made conditional upon both the record status and the requested transaction.

Figure 2 shows the unmodified code for the primary record search subroutine; Figure 3 shows the code for the modified subroutine. Additions to the search subroutine are in boldface type in Figure 3: the command CMD is passed into the search subroutine and the conditions for successful search are altered. With the modifications shown in Figure 3, records are "found" in the data base under either of two conditions:

1. The Command (CMD) is RESTORE, a record with the correct search key is found, and the status of the record that is found is "deleted."
2. The command (CMD) is *not* RESTORE, a record with the correct search key is found, and the status of the record that is found is "active."

While the solutions to the RESTORE enhancement task varied somewhat from the prototypical solution illustrated in Figure 3, most enhancements were in the spirt of the solution shown here. (See Soloway et al [2] for a discussion of the quality of this enhancement.)

4. Building Mental Model of Programs

We call the knowledge programmers acquire about a program a mental model of the program. We define *weak mental models* of a program to be those mental models that contain only static knowledge of the program. We define *strong mental models* of a program to be mental models that contain both static knowledge and causal knowledge about it.

We observed that strong and weak mental models result from using different strategies for studying the PDB program. Some programmers used a systematic strategy for studying the program, whereas other programmers used an as-needed strategy. Programmers using the systematic strategy to study the PDB program gathered both static and causal knowledge about it and so built strong mental models of the program. Programmers using the as-needed strategy gathered only static knowledge of the program and therefore built weak mental models.

Figure 2. Unmodified search subroutine.

```
SUBROUTINE SEARCH(dbase, iptr, name)
DO 700 i = 1, ifinal
  IF (name(1: ipos-1) .EQ. dbase(i, 1)(1: ipos-1)  COMMENT 1
     .AND. dbase(1, 7) .EQ. 'active')              COMMENT 2
  THEN iptr = i                                    COMMENT 3
. . .
END
COMMENT 1: Compare names
COMMENT 2: Be sure record 'active'
COMMENT 3: Return pointer to record
```

In Section 5 we show that the programmer's use of a strategy to study the program leads directly to the knowledge he or she acquires about the program. In order to illustrate this finding, we present anecdotal excerpts from videotaped interviews with two programmers. The interviews were conducted while the programmers studied the PDB program and made their RESTORE enhancements. We use the excerpts to illustrate, compare, and contrast the systematic and as-needed strategies for studying programs and the different knowledge that results from using each strategy.

One of the programmers, SYS1, used the systematic strategy; the second programmer, A-N1, used the as-needed strategy. We present a comparison of these two programmers for three reasons:

1. SYS1's and A-N1's study strategies are at opposite ends of a continuum: SYS1 studied the PDB very systematically; A-N1 studied the program using an as-needed strategy.
2. SYS1's RESTORE enhancement was prototypical of successful modifications: By passing a parameter to the record search subroutines indicating whether the user requested the RESTORE transaction, SYS1 forced the record search subroutines to return the "deleted" records RESTORE required. Nearly every successful RESTORE enhancement used this method to provide "deleted" records to RESTORE.
3. The way in which A-N1's RESTORE enhancement failed was prototypical of programmers who used the as-needed strategy: A-N1 did not adjust the record search subroutines to locate "deleted" personnel records when the user requested the RESTORE transaction. Therefore, A-N1's record search subroutines could not find the "deleted" personnel records as RESTORE requires.

SYS1, the programmer who used a systematic strategy, acquired the static and causal facts about the program, built a strong mental model, and constructed a successful RESTORE enhancement. A-N1, who used the as-needed strategy, learned only the static facts, built a weak mental model, and failed to construct a successful RESTORE enhancement.

5. THE SYSTEMATIC AND AS-NEEDED STRATEGIES COMPARED

A programmer who uses either the systematic strategy or the as-needed strategy to study a program acquires knowledge about the program. However, the knowledge the programmer acquires depends on which of the two strategies is used to study the program. The two strategies focus on gathering different kinds of knowledge about the program. Programmers using the as-needed strategy focus on local facts about the program. In contrast, programmers using the systematic strategy are concerned not only about the local facts, but also

Figure 3. Enhanced search subroutine.

```
SUBROUTINE SEARCH(dbase, iptr, name, CMD)
DO 700 i = 1, ifinal
  IF (name(1: ipos-1) .EQ. dbase(i, 1)(1: ipos-1)              COMMENT 1
   .AND.
       ((CMD .NEQ. 'r' .AND. dbase(1,7) .EQ. 'active')         COMMENT 2
    .OR.
      (CMD .EQ. 'r' .AND. dbase(1,7) .EQ. 'deleted'))          COMMENT 3
  THEN iptr = i                                                COMMENT 4
. . .
END
COMMENT 1: Compare names
COMMENT 2: Be sure command is not a RESTORE and record is 'active'
COMMENT 3: Be sure command is a RESTORE and record is 'deleted'
COMMENT 4: Return pointer to record
```

about discovering relationships among parts of the program. In what follows we examine three issues that are central to the two study strategies:

1. What are the programmers' goals in studying the program?
2. Which functional components does the programmer choose to study?
3. Does the programmer study the interactions of the program's components?

The way in which these three questions are answered determines whether the programmer uses the systematic or the as-needed strategy. In this section, we present examples from interviews with SYS1 and A-N1 that illustrate how programmers using the systematic or the as-needed strategy resolve these three issues, resulting in the acquisition of either static *and* causal information about the program or only static information.

5.1. Programmers' Goals

Programmers' assessments of their need to understand the PDB program differed; some believed that they needed a thorough understanding of the program, whereas others believed that a partial understanding would be sufficient. Their study strategies reflected the extent to which they wanted to understand the program. Programmers who judged that in order to make a correct enhancement to the program, it would be necessary to understand how the behavior of the enhancement would interact with the behavior of the rest of the program used the systematic strategy to understand the PDB program. As a result of their judgment, they wanted to understand the interactions of the components of the PDB program as it executes. In contrast, programmers who used the as-needed strategy did not believe that they had to understand the interactions in the program in order to make their enhancements.

5.1.1. Systematic Strategy: SYS1. Understanding the unmodified PDB program was very important to SYS1. Before starting to implement his RESTORE enhancement, he studied the program extensively. The interviewer asked him why he did this. We have paraphrased SYS1's response here.[1]

SYS1: The easiest way for me to work on a new project is to see how the original programmers were doing it ... and carry on with their way, rather than try to work the whole thing my way because my way may not mesh with their way. Then we'd have a problem when I put my module or my enhancement in—it might not fit. If I see how they've done things first, then I can make my enhancement fit.

SYS1's remarks illustrate his belief that by understanding how the unmodified program is organized and functions, he would be able to construct a RESTORE enhancement whose behavior would mesh with the rest of the PDB program. SYS1 believed that by understanding how the original program is organized, and how it functions, he could construct an enhancement that would minimize the likelihood of disrupting the functioning of the program.

5.1.2. As-Needed Strategy: A-N1. A-N1 believed that he did not have to understand the whole PDB program to construct his RESTORE enhancement. A-N1 justified his satisfaction with a partial understanding of the program by observing that the PDB program is quite small and therefore unlikely to involve any complex interactions. We have paraphrased his remarks here.

A-N1: Since this system is small I think I would feel good about modifying the program without understanding it. But, if this were a large, really complex system, I really think I would want to understand it better, since there might be some subtle little problem that might exist in the interactions of the subroutines.

A-N1 realized that in large programs, subtle, unanticipated problems could emerge as a result of modification. The PDB program is evidently small enough, and constructed simply enough, that A-N1 did not feel it necessary to understand it in detail. Though A-N1 recognized that he did not understand the program in detail, he nonetheless believed that his RESTORE enhancement would function correctly with the rest of the program:

A-N1: I'm 90% sure the thing [the RESTORE enhancement] would wcrk perfectly, but I wouldn't understand it [how the enhanced program worked].

These views illustrate A-N1's belief that since the PDB program is small, he could make changes to it without understanding how the program functions as a whole. In contrast, SYS1 wanted to understand the program in order to ensure that his RESTORE enhancement would be coordinated, or mesh, correctly with the original program. In Sections 5.2 and 5.3 we describe and compare the processes by which the programmers SYS1 and A-N1 attempted to collect static and causal information about the PDB program. Following each description of the programmer's information-gathering strategy, we

[1] As a result of stuttering, faltering, and reorganizing thoughts as they speak, our subjects' protocols tend to include spurious expressions, repeated phrases and words, and segments of false-start sentences. These fragmented expressions make reading the protocols difficult. This segment of the interview, though very revealing, was long and difficult to read, so we have paraphrased it, capturing the intent, omitting only the extra words.

provide evidence from verbal protocols that the programmer succeeded or failed to acquire the information necessary to correctly implement the RESTORE enhancement.

5.2. Acquiring Static Knowledge

In order to understand a program, a programmer must identify the program's functional components. We define a functional component of a program to be segments of code that, together, accomplish a task. In the PDB program many functional components correspond to single subroutines or groups of subroutines. By identifying the functional components of the PDB program, programmers gather and organize static facts about the program. In Section 5.2.1 we describe the process by which SYS1 gathered static knowledge and the knowledge he acquired. Then, in Section 5.2.2 we show that A-N1 gathered information by a similar process and also gathered the static knowledge essential to performing the RESTORE enhancement.

5.2.1. Systematic Strategy: SYS1. Before beginning his RESTORE enhancement, SYS1 examined most of the PDB program, identifying functional components and their organization. Beginning at the top of the main routine, SYS1 read through the code, noting the ordering of the subroutine calls. Initially inferring the function of each subroutine from its name, SYS1 linked these functions together in the order in which they occur as the program executes, thus developing an understanding of the global data flow in the program. After this initial study of the PDB main routine, SYS1 returned to each subroutine call, in order, and examined the code associated with it. Thus, as SYS1 studied the organization of the subroutine calls in the main routine of the program, SYS1 also identified all the functional components that were necessary to achieve his goal of understanding the program. Through this process, SYS1 acquired the static knowledge necessary to perform the RESTORE enhancement correctly.

1. Each record contains an activity status field. Before he studied the PDB program in detail, SYS1 considered how a RESTORE enhancement might function. SYS1 generated several possibilities for the RESTORE enhancement and appeared to favor one that somehow kept track of records deleted during each session so that they could be restored, if necessary, during the session. To keep track of records that had been deleted during a session, SYS1 entertained the possibility of adding a status field to each personnel record that would store the information about whether the record had been deleted during the session or was still active. When SYS1 began to study the program in detail he discovered that, in fact, each personnel record already does contain an activity status field.

   SYS1: It [the PDB program] keeps track of the records, marks them "active" or "deleted."

2. DELETE changes the status field to deleted. With the knowledge that each record contains an activity status field, SYS1 studied the DELETE subroutine to determine how the status field is changed to cause deletions. When SYS1 saw how DELETE performs deletions of personnel records, he observed:

   SYS1: It [DELETE] marks the activity status as "deleted."

3. Records are never physically deleted. Since the RESTORE enhancement would have to restore "deleted" records, SYS1 wanted to know whether the RESTORE enhancement would have access to all the records deleted in a session with the PDB program. Considering the purpose of the activity status field, SYS1 generated the hypothesis that records are not physically deleted from the personnel data base. Rather, deletions are "virtual deletions" that leave the records physically in the data base but somehow make them "invisible" to the transaction subroutines. This is how he stated that hypothesis, which he subsequently confirmed:

   SYS1: I'm thinking that the record isn't actually deleted.

4. Record search bypasses records marked "deleted." After SYS1 discovered that deletion of a record from the personnel data base is not accomplished by physically deleting the record, but rather by changing its status from "active" to "deleted," he discovered that they search subroutines find only "active" records:

   SYS1: But it [the record] won't "exist" ... they [the record search subroutines] check if status is "active" when they search for it.

Thus by examining the code in execution order, SYS1 acquired all four of the critical pieces of static knowledge about how the PDB program deletes and selects personnel records according to activity status. A-N1 also collected all these pieces of static knowledge, although using a different strategy.

5.2.2. As-Needed Strategy: A-N1. Like SYS1, A-N1 began his study of the PDB program by identifying some of the program's functional components. Unlike SYS1, however, A-N1 wanted to learn only as much about the program as he believed would be necessary to accomplish the RESTORE enhancement. As a result, after reading the enhancement assignment, and reasoning that RESTORE should simply reverse the action of some hypothetical record deletion functional component, A-N1 wanted to understand this deletion function. At this point, A-N1 remarked to the interviewer that he hoped that the program was modularly organized, specifically that the record deletion functional component corresponded to a DELETE subroutine. A-N1 looked at the documentation for the program's main routine; in the documentation of the main routine, A-N1 found a list of the subroutines called by the main routine, the subroutine DELETE among them. In his remarks, A-N1 made it clear that he had now succeeded in identifying the only functional component with which he was immediately concerned, namely, DELETE.

A-N1: It looks like they may have broken this [program] up functionally. Ah, [here are the] subprograms called [by the main routine]: CREATE, DELETE, GETCMD ... Ok, good. So, there is a subprogram called DELETE. So, I think I'm going to go straight to that.

A-N1's sentiment, expressed in this quotation, matches his behavior perfectly: He went directly to the DELETE module, studied it, and began his RESTORE enhancement. He did not look at any other part of the PDB program before beginning his RESTORE enhancement.

In the following excerpts from the interview with A-N1, we show that A-N1, like SYS1, had the critical static knowledge about the PDB program.

1. Each record contains an activity status field. Though A-N1 had not studied the declaration of the array that contained the data base, he inferred that references to the variable DBASE were references to the data base array. Thus, when A-N1 read the DELETE subroutine and discovered that it changes the last field in each record of DBASE from "active" to "deleted," he reasoned that each record contains an activity status field.

   A-N1: The status of the record, as to whether it was "deleted" or not "deleted"—it's contained in the record itself.

2. DELETE changes status field to "deleted." When A-N1 studied the DELETE subroutine, he was attempting to discover whether his RESTORE enhancement could simply reverse the action of DELETE. In order for RESTORE simply to reverse the action of DELETE, records must not be physically deleted from the PDB. The following quotation shows A-N1 discovering that the record deletion mechanism alters the value of the record activity status field.:

   A-N1: Deletion consisted of a one line statement [in the DELETE subroutine] that just said that the contents of [the record status field] ... would be the ASCII string 'deleted.'

3. Records are never physically deleted. Once A-N1 discovered that deletion of a record from the PDB is accomplished by changing the value in the record's activity status field, he needed to know whether the "deleted" record would be available for subsequent use or whether it would be physically deleted because it contained "deleted" in the activity status field. The following quotation shows that he did indeed discover that records are never physically deleted from the PDB.

   A-N1: Deleted records are never really deleted, so that in the process [of writing the data base back to the disk], they too are rewritten.

   The fact that personnel records are never physically deleted from the data base was crucial for the design of A-N1's RESTORE enhancement. A-N1 concluded, from the facts that records are simply marked "deleted" and that records are never physically deleted, that RESTORE could indeed be a "mirror image" of DELETE.

4. Record search bypasses records marked "deleted." After completing his RESTORE enhancement, A-N1 began to investigate the rest of the program. He explored the program not because he felt his RESTORE enhancement was incorrect, but because he wanted to verify his assumption that RESTORE was the mirror image of DELETE. A-N1 recognized that the RESTORE routine would have to obtain records from another component of the program, so he wanted to learn about the record retrieval process. He found the record search process and read the code. At the end of the interview, the interviewer asked A-N1 to describe how the PDB program functioned. Among his remarks was this evidence that he understood that the search overlooked "deleted" records:

   A-N1: If it [the search process] doesn't find the word "active" then it returns "record not found" or some other message.

In summary, A-N1 and SYS1 used some of the same reasoning to identify the program's functional components. Both expected that the program was modularly organized and that the names of the subroutines would be mnemonic and would reflect their function. A-N1 and SYS1, however, had different goals in studying the program. SYS1's desire to understand the whole program led him to identify almost all of the program's functional components before beginning his RESTORE enhancement. In contrast, A-N1's desire to understand as little as necessary about the PDB program led him to identify only one functional component before he began his RESTORE enhancement. A-N1 studied only DELETE before attempting to construct his RESTORE enhancement because, according to A-N1, the action of RESTORE most closely parallels the action of DELETE.

Identification of the functional components of the PDB program provides programmers with static facts about the program. Knowing the static facts about the functional components of the program is necessary, but not sufficient, for constructing a successful RESTORE enhancement. Though both programmers learned the four critical pieces of static knowledge, SYS1 acquired additional static knowledge that was useful when he attempted to identify interactions between actions and functional components. As we explain in Section 5.3, correctly integrating the RESTORE enhancement into the PDB program requires understanding how the functional components of the program causally interact through data flow when the program executes. Thus, in order to integrate the RESTORE subroutine correctly into the original program, a programmer must also have causal knowledge about the interactions of the running program's functional components. By examining the order in which functional components are called in the running program, SYS1 both identified the functional components, by using the mnemonic values of their names, and learned which functional components depend on the results of other components to satisfy their preconditions. This knowledge of the data flow and run-time ordering of the program's functional components gave SYS1 a framework for how the components interact as the program runs. Though A-N1 used a similar method for identifying functional components, namely, inferring a subroutine's function from its name, A-N1's desire to begin his RESTORE enhancement immediately led him to examine only the DELETE subroutine before implementing his RESTORE enhancement: A-N1 thus neither (1) identified all the functional components in the program, nor (2) acquired knowledge about the causal interactions of the program's functional components before he began implementing his RESTORE enhancement. In the next section, we describe SYS1's use of global symbolic execution to acquire causal information about interactions in the running program; we also show the contrast to A-N1's strategy of gathering information about the program only when that information is necessary.

5.3. Acquiring Causal Knowledge

Though both SYS1 and A-N1 identified the functional components containing the four pieces of static knowledge essential to performing the RESTORE enhancement correctly, they did not both determine how those functional components interact with one another when the program runs: SYS1 used global symbolic execution to detect the interaction between the record search process and the RESTORE subroutine's need for "deleted" records; A-N1, prior to implementing his RESTORE enhancement, examined the code only of the DELETE subroutine. He did not perform global symbol execution, but rather immediately focused on the code local to the DELETE subroutine in order to limit his study of the program.

5.3.1. Systematic Strategy: SYS1. Like all programmers who used the systematic strategy to study the PDB program, SYS1 attempted to determine how the PDB program behaves when it runs. SYS1's desire to understand how the program behaves when it runs led directly to the goal of determining how the functional components interact when the PDB program runs. In order to determine how functional components of the program interact when it runs, programmers using the systematic strategy used extensive global symbolic execution. Global symbolic execution entails symbolically executing the program, beginning at the main routine and following the data flow and calling structure of the subroutines. Because global symbolic execution requires that the programmer actually imagine the behavior of the program as if it were running in time, global symbolic execution provides the programmer with data flow knowledge, and therefore causal knowledge, about the order of actions in the program. Knowledge of the data flow and the order of actions in the program permits the programmer to reason about precondition relationships that lead to interactions.

SYS1 began global symbolic execution of the program at the main routine, symbolically executing each line of code in the main program in the order in which it is executed at run time. In his initial symbolic execution, SYS1 used a breadth-first strategy: He postponed symbolically executing the called subroutines until after he understood the behavior of the main routine. SYS1 explained his performing global symbolic execution of the program by saying, "I'm tracing through the main program to see the flow of the main program through its subroutines." SYS1 uses the term *flow* to refer to the causal sequencing of events in the PDB program, i.e., data flow. The statement therefore shows that SYS1 used global symbolic execution to discover how causally dependent actions of the running program are ordered. Ordering the actions of the functional components of the program makes it possible for the programmer to detect when functional components interact causally with one another. Detection of such interactions permits the programmer to mesh the modification with the existing program.

RESTORE interacts with record search. Based on his static knowledge of the PDB program, SYS1 understood that the record search subroutines find "active" records only. From his symbolic execution of the program's main routine, SYS1 also had the causal knowledge about data flow that each transaction subroutine is called at the same point in the PDB program. SYS1 realized that if the record search process were carried out *before* the transaction subroutine were called, then the same record search process would be used for all transactions. As a result, SYS1 realized that he would have to alter the PDB program to enable the record search process to find "deleted" records when the user requested the RESTORE transaction. The following quotation shows that SYS1 was fully aware of the potential for this interaction between RESTORE and the record search process.

SYS1: If it's a RESTORE, I actually don't want "active" status. So if that [whether record is active] was checked in a certain place, like before I got the command, then I'll have to rearrange some stuff."

This piece of causal knowledge provided the key for SYS1's RESTORE enhancement. Since the PDB program carries out the record search process before, and therefore independently of, the transaction routines *and* returns only "active" records, SYS1 recognized that RESTORE could not obtain "deleted" records. SYS1 expressed the problem this way:

SYS1: Somehow, we have to tell SEARCH [the search process] not to look for an "active" record.

In order to "tell" the search process not to look for an "active" record, SYS1 capitalized on the fact that the PDB program asks for the transaction command before carrying out the record search: His solution used the command itself to inform the search subroutines whether they should look for "deleted" records or "active" records. If RESTORE were the transaction the user requested, then the search process would return "deleted" records; if any other transaction were requested, then the search process would return "active" records. SYS1 characterized his solution in this way:

SYS1: We know the command at that point. So, what I would do is put a check at this point [in the search subroutines] to check if the command is RESTORE.

Thus, SYS1 recognized the causal interaction of the record search process with RESTORE's need for "deleted" records and invented a method for accommodating it. By informing the record search subroutines of the name of the transaction the user requested, SYS1 made it possible for the record search subroutines to return "active" or "deleted" records conditional upon the requested transaction.

In summary, SYS1's systematic strategy for studying the PDB program led him to acquire critical static and causal knowledge and thus to build a strong mental model of the program. The critical static knowledge allowed SYS1 to reason about how the program is constructed. The critical causal knowledge allowed SYS1 to reason about how the components of the program interact causally through data flow when the program runs. Thus, the combination of the critical static and causal knowledge made it possible for SYS1 to construct a RESTORE enhancement that functioned correctly with the original PDB program.

5.3.2. As-Needed Strategy: A-N1. Immediately after reading the RESTORE assignment, A-N1 tried to find a local segment of code he could change to implement the RESTORE enhancement. Rather than examine the whole PDB program, A-N1 briefly read the overview of the program, and upon learning that a DELETE module exists, he reasoned that RESTORE would simply have to reverse the action of DELETE. When A-N1 reasoned in this way, he had not yet examined any code and he did not know how records were deleted in the original program. A-N1 then skimmed over the documentation of the program's main routine, confirming his expectation that each of the data base transactions mentioned in the program overview corresponds to a subroutine in the program. A-N1 then turned immediately to the code and documentation for the DELETE module, read them, and began his modification. A-N1 verbally expressed his general strategy of trying to start a modification task immediately, without understanding how the program functions, in the following quotation:

A-N1: It's best to go straight to the thing where you think you'll find a solution. And if that doesn't work, backtrack and find out where you went wrong.

A-N1 followed this prescription exactly. A-N1 used the definition of the RESTORE enhancement to identify sections of the PDB program that might provide him with information that would allow him to make local changes to the program to implement RESTORE. A-N1 believed that this section of code would be a record deletion subroutine. A-N1 searched for the part of the program he expected to carry out the record deletion function, found that section in the DELETE subroutine, and then immediately attempted to construct a modification of the DELETE subroutine to implement the RESTORE enhancement. When A-N1 needed to know something in order to proceed with the enhancement, be backtracked to fill in missing knowledge, just as his quotation suggests. A-N1's study of the program was thus guided by his need to collect information directly relevant to the enhancement. Rather than being sure that RESTORE would fit in with the rest of the program, A-N1 adopted the strategy of modifying DELETE in the following way:

A-N1: All I need to do is take this code [DELETE] verbatim, call it RESTORE, and put the word "active" in it instead of "deleted."

This quotation shows that A-N1 believed that RESTORE was nothing more than a "mirror image" of DELETE, require only local modifications of the PDB program. It is especially important to realize that A-N1 never verified his assumptions that local modifications would not disrupt the data flow in the program and that the existing data flow would be sufficient to implement a correct RESTORE enhancement. At the interviewer's prompting, A-N1 examined more of the PDB program after implementing his RESTORE enhancement. As A-N1 examined the program at that time, he discovered several facts that might have led to understanding shy his RESTORE enhancement was incorrect. As we will show in Section 5.3.3, however, since A-N1 had a weak mental model of the PDB program, he was unable to see how these facts interacted with his RESTORE enhancement.

A-N1's static knowledge was sufficient for him to attempt to construct a RESTOR subroutine, though the solution was incorrect. His solution, described in the last quotation, implements the RESTORE subroutine by making only local changes to the PDB program. A-N1 believed that RESTORE could be the mirror image of DELETE. A-N1's RESTORE enhancement was based on the assumption that the interaction of RESTORE with the rest of the PDB program would be exactly like the interaction of DELETE with the rest of the PDB program. This assumption, that RESTORE and DELETE have identical requirements of the PDB program, is fallacious. The assumption led to the failure of his RESTORE enhancement: Although A-N1 was able to construct a RESTORE subroutine, he was *not* able to integrate his subroutine into the original program. As we demonstrate in what follows, A-N1 failed to acquire the causal knowledge necessary to understand how RESTORE and the record search process interact in the executing program. This demonstration suggests that knowledge of the interaction between RESTORE and the record search process was critical to integrating the subroutine into the PDB program.

RESTORE interacts with record search. In the debriefing phase of the interview, the interviewer asked A-N1 a series of questions about how the PDB program, enhanced with A-N1's RESTORE subroutine, behaved when it was looking for "deleted" records. A-N1's answer to the interviewer's key question shows graphically that A-N1 was completely unaware of the interaction of RESTORE and the order of actions in the PDB program.

INTERVIEWER: When you write a RESTORE routine to restore records after DELETE deleted them, how does SEARCH [the search subroutine] find them [the "deleted" records]?

A-N1: Well, the RESTORE routine puts the "active" string in the status [field]. So when SEARCH finds it and compares it to the ASCII string "active," ... [SEARCH] says, "Oh, it's here," and returns the pointer back.

We know from A-N1's earlier remarks that he had the critical static knowledge about the PDB program required to construct a successful RESTORE enhancement. In this quotation, however, A-N1 showed that he did not know how his enhancement to the PDB program would interact with the record search process. He suggested that RESTORE itself solves the problem of only being able to find "active" records: According to A-N1, RESTORE itself changes the status of "deleted" records to "active" so that the record search subroutines can find them! Apparently, A-N1 did not know that RESTORE depends on the record search process to find a record *before* RESTORE can restore the record to active status. Hence, A-N1 did not have the key piece of causal knowledge required to construct a successful RESTORE enhancement.

Unlike SYS1, A-N1 never attempted symbolic execution of the main routine of the PDB program to discover causal interactions among subroutines. A-N1's use of the as-needed strategy led him to focus on local aspects of the program rather than on its global behavior. In essence, A-N1's attention to the program was guided by his need to gather information required to answer specific questions about local program behavior that arose during the process of performing the RESTORE modification. Since A-N1 focused on local behavior of the PDB program, he was less likely to put together the hints at global interactions that were present in both the descriptions of the subroutines and the actual code. SYS1's use of the systematic strategy and his concern with understanding the global behavior of the program before attempting to modify it led him to acquire both static and causal knowledge about the program; A-N1's focus on local aspects of the program led him to acquire only static knowledge and to overlook the causal knowledge. Since A-N1 did not understand how the PDB program behaved when it executes, he failed to acquire the causal knowledge necessary to understand the way in which the RESTORE enhancement interacts with the existing PDB program.

The next section of this paper examines the statistical relationship between systematic study of the program, extensive symbolic execution of the code, and success with the RESTORE enhancement task. We show that our anecdotal descriptions of SYS1 and A-N1, which portray how strong and weak mental models are built and used, are strongly supported by statistical evidence linking systematic study and success in making the RESTORE enhancement to the PDB program. The crucial feature of this link between the systematic approach and success appears to be the use of symbolic execution.

6. STATISTICAL EVALUATION OF SUCCESSFUL SUBJECTS

Just as SYS1's behavior led to a successful RESTORE enhancement, programmers who produced successful RESTORE enhancements behaved differently from unsuccessful programmers like A-N1. Successful programmers acquired different knowledge about the PDB program as well. Programmers who designed successful RESTORE enhancements uniformly took a systematic approach to studying the PDB program. They attempted to learn how the program was constructed and how it functioned when it ran. This knowledge about how the program functioned was evidently acquired through extensive symbolic execution. The successful systematic programmers also had critical knowledge that the unsuccessful programmers lacked: They knew how the process of searching for a record interacted with the RESTORE enhancement, and they realized that searching for a record occurred before it was processed. Thus, successful programmers

used a predominantly systematic approach to studying the program;

used extensive symbolic execution when studying the PDB program;

understood the interaction between the record search subroutines and RESTORE.

Table 1 shows how success was associated with systematicity, symbolic execution, and awareness of the interaction of RESTORE and the record search process. Ten programmers, five of whom were successful and five of whom were not, are represented. Every successful subject used a systematic approach, performed extensive symbolic execution, and understood the interaction of RESTORE and the process of searching for a record. The predominant strategy column, for example, shows that the use of a systematic strategy permitted all five subjects who used it to design successful RESTORE enhancements; all five programmers who used the as-needed strategy failed. The next two columns show, respectively, that extensive symbolic execution of the PDB code and understanding the interaction of SEARCH2 and RESTORE were also perfectly associated with success.

Each of these relationships was statistically significant, using chi-square analyses and significance levels corrected for performing multiple tests on the same subjects. Each of the chi-squares relating success and one of the three variables (i.e., using the systematic strategy, performing extensive symbolic execution, and

Table 1. What Successful Maintainers Did and Knew

	Predominant Strategy		Extensive Symbolic Execution		Understood Interaction of Record Search and RESTORE	
Result	Systematic	As-needed	Yes	No	Yes	No
Success	5	0	5	0	5	0
Failure	0	5	0	5	0	5

n = 10.

understanding the interaction of the record search process and RESTORE) was significant beyond the 5% level. We conclude from these analyses that the likelihood of successfully designing a RESTORE enhancement is greatly increased if programmers are systematic in their approach to studying the program and perform extensive symbolic execution. Finally, we note that there was virtually no relationship between years of professional programming experience and either successfully performing the enhancement task or the programmer's choice of study strategy. The point-biserial correlation between success in the enhancement task and years of professional programming experience was −0.06, which does not approach significance. This value of the correlation coefficient is effectively 0. Since success in the enhancement task corresponded exactly to the study strategy the programmer used, this correlation coefficient shows that there is no relationship between experience and the programmer's choice of study strategy.

7. DISCUSSION AND CONCLUSIONS

Our empirical results demonstrate that programmers who used the systematic strategy were able to construct successful RESTORE enhancements for the PDB program, whereas programmers who used the as-needed strategy were unable to do so. We observed that programmers who used the systematic strategy performed extensive global symbolic execution of the PDB program; programmers who performed the as-needed strategy did not use extensive symbolic execution. We also observed that although all programmers acquired the necessary static knowledge about the program, only programmers who used the systematic strategy acquired the causal knowledge necessary to make the RESTORE enhancement successfully. Thus, it appears that the use of global symbolic execution provides programmers who used the systematic strategy with necessary causal knowledge that is required to construct successful RESTORE enhancements.

The connection between global symbolic execution and the acquisition of causal knowledge can be understood by considering the role of knowledge about data flow in understanding programs. Successfully modifying a program requires being able to determine how proposed modifications to the code affect the data flow between program components. The programmer must be able to reason about the program's data flow in order to assess whether proposed modifications to the code would disrupt the fulfillment of preconditions, or if a proposed modification requires modifying other parts of the program to preserve the fulfillment of preconditions. By performing extensive symbolic execution of the PDB program, the programmer who uses the systematic strategy simulates the code in detail, imagines its behavior at run time, and determines how to map the program's behavior onto the program's code. Detailed simulation of the code thus forces the programmer to imagine the data flow across components of the program; communication across components of a program is, by definition, the manner in which causal interactions among components are achieved. Understanding the data flow in the unmodified program permits the programmer to reason about what the data flow should be in the modified program. Programmers who used extensive global symbolic execution to study the program understood the data flow among the components of the PDB program; that knowledge permitted them to design RESTORE enhancements that avoided destructive interactions.

In contrast to programmers who used the systematic strategy, programmers who used the as-needed strategy to understand the PDB program wanted to minimize their understanding of the parts of the PDB program that they believed were tangential to their enhancements. Programmers who used the as-needed strategy gathered information about the program only when they needed it to resolve questions that arose as they tried to implement their enhancements. As a result, programmers who used the as-needed strategy did not perform extensive symbolic execution of the PDB program and therefore did not understand the data flow in the program. Since programmers who used the as-needed strategy did not understand the data flow in the program, they were unable to reason about how the components of the program interact when the program runs; thus they could not construct enhancements that avoided the destructive impact of the interactions.

The current research methodology confronted professional program maintainers with a situation and task that differed in two significant ways from their normal experience. Further research must address the effects of these two important differences.

What role does the test and debug cycle play in program maintenance?

What study strategies are appropriate for large programs?

Subjects in our study were not allowed to test and debug their enhancements. Testing and debugging a program bring to the programmer's attention information that was overlooked in the initial study of the program. Programmers using the as-needed strategy may depend more heavily on testing and debugging to learn about the program's structure than programmers using the systematic strategy. If we had permitted a test and debug cycle, the programmers who used the as-needed strategy might have constructed successful RESTORE enhancements. If programmers using the as-needed strategy with the test and debug cycle could correctly implement the RESTORE enhancement, they would at least have to acquire the necessary causal knowledge identified in Section 3. If the programmers in our study had been allowed to test and debug their enhancements, it is possible that the mental models resulting from both strategies would have been equally strong. What, then, would recommend one study strategy over another?

Decisions about which study strategy is "best" should take into account the effects on programming efficiency of the strength of the mental model. If the mental models resulting from the use of the two study strategies were equally strong, then the programmer should use the most efficient study strategy. The most efficient study strategy may not, however, be the fastest or the one that produces immediate results. In a study of the efficiency of the software development process, Fagan [3] presents data suggesting that fixing errors as early as possible in the software development process is more efficient than fixing errors late in the development process. This suggests that finding and correcting errors in the design of an enhancement before implementing the enhancement is most efficient. By understanding the program prior to implementing their enhancement, programmers using the systematic strategy may be able to avoid or detect errors in the design of an enhancement more efficiently than programmers using the as-needed strategy. Programmers using the as-needed strategy may have to depend on testing and debugging to correct errors in the design of the enhancement after the enhancement has been implemented. Though their initial study of the program and construction of an enhancement may be faster, they would incur the added expense of tracking down and eliminating the effects of their early, undetected mistakes. On the other hand, if the mental models resulting from the use of the two study strategies were not equally strong, then the programmer would have to consider the amount of time spent to complete the initial enhancement, the amount of time spent and debug that enhancement, and the value to future maintenance of having a clear understanding of the program. The impact of a clear understanding of the program would be felt in two ways. Subsequent maintenance tasks depend on (1) the programmer's ability to debug the program and (2) the continuing integrity of the modified program. If the programmer's ability to debug proposed solutions and integrate them cleanly into the existing program depend on the strength of the mental model, then the programmer should use the study strategy that results in the strongest mental model. Since the current research does not address the impact of the test and debug cycle on the strength of the resulting mental model, this is an important area for future investigation.

The PDB program contains 250 lines. Clearly, it is not possible to use the systematic strategy to study a program of 100,000 lines in preparation for modifying it. On the other hand, the current research strongly suggests that causal knowledge provides a powerful basis for performing program maintenance tasks. How, then, can we assist program maintainers working with large programs in constructing what we have labeled strong mental models of large programs? Since the systematic strategy is unworkable for studying large programs, programmers working with large programs will be forced to use study methods akin to the as-needed strategy. In order to assist maintainers faced with large programs, it will be necessary to augment their as-needed program study methods. However, the as-needed strategy appears to consist of a much larger, more loosely organized collection of methods that we only partially understand. Thus, further investigation of the as-needed strategy would be a productive approach to defining the ways in which the selective study of large programs can be combined with automated study tools that assist programmers in developing powerful causal models of programs. We may begin such a study by identifying the methods called on by programmers using the as-needed strategy and by identifying the knowledge that results from various components of the as-needed strategy and how to supplement it.

Our concerns about the relative efficacy of the systematic and as-needed study strategies, the role of testing and debugging in program understanding and development, and the problems of understanding large programs are largely concerns about the specifics of study strategies. Our research shows that programmers who use the systematic strategy to study the PDB program acquire causal knowledge about programs. It is the importance of causal knowledge in program understanding that constitutes the main contribution of our work, not the description of the study strategies themselves. We hope that the systematic strategy is not the only way to acquire causal knowledge of a program: The systematic strategy is clearly unworkable for understanding large programs. Now that we understand the importance of causal knowledge for program understanding, further research must investigate other study strategies and develop programming tools to help programmers acquire the causal knowledge necessary to build strong mental models of programs.

ACKNOWLEDGMENT

We express our thanks to James Sophrer and the other members of the Cognition and Programming Project for their comments and support.

REFERENCES

1. R. K. Fjeldstad and W. T. Hamlen, Application program maintenance study—Report to our respondents, in Tutorial on Software Maintenance. (G. Parikh and N. Zvegintzov, eds.). Silver Spring, MD: IEEE Computer Society Press, 1983.
2. E. Soloway, S. Letovsky, B. Loerinc, and A. Zygielbaum, The cognitive connection: Software maintenance and documentation, *Proc. 9th Annual Workshop on Software Engineering,* pp. - , Atlanta, GA, 1984.
3. M. E. Fagan, Design and code inspections to reduce errors in program development, *IBM Systems J.* 15(3):182–211 (1976).